2013

Improving Security and Privacy in Online Social Networks

Wei Wei
College of William & Mary - Arts & Sciences

Improving Security and Privacy in Online Social Networks

Wei Wei

Beijing, China

Master of Science, Beihang University
Bachelor of Science, Beihang University

A Dissertation presented to the Graduate Faculty
of the College of William and Mary in Candidacy for the Degree of
Doctor of Philosophy

Department of Computer Science

The College of William and Mary
August 2013

APPROVAL PAGE

This dissertation is submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

Wei Wei

Wei Wei

Approved by the Committee, July, 2013

Committee Chair
Associate Professor Qun Li, Computer Science
The College of William and Mary

Professor Weizhen Mao, Computer Science
The College of William and Mary

Associate Professor Haining Wang, Computer Science
The College of William and Mary

Associate Professor Gang Zhou, Computer Science
The College of William and Mary

Associate Professor Gexin Yu, Mathematics
The College of William and Mary

ABSTRACT

Online social networks (OSNs) have gained soaring popularity and are among the most popular sites on the Web. With OSNs, users around the world establish and strengthen connections by sharing thoughts, activities, photos, locations, and other personal information. However, the immense popularity of OSNs also raises significant security and privacy concerns. Storing millions of users' private information and their social connections, OSNs are susceptible to becoming the target of various attacks. In addition, user privacy will be compromised if the private data collected by OSNs are abused, inadvertently leaked, or under the control of adversaries. As a result, the tension between the value of joining OSNs and the security and privacy risks is rising.

To make OSNs more secure and privacy-preserving, our work follow a bottom-up approach. OSNs are composed of three components, the infrastructure layer, the function layer, and the user data stored on OSNs. For each component of OSNs, in this dissertation, we analyze and address a representative security/privacy issue. Starting from the infrastructure layer of OSNs, we first consider how to improve the reliability of OSN infrastructures, and we propose Fast Mencius, a crash-fault tolerant state machine replication protocol that has low latency and high throughput in wide-area networks. For the function layer of OSNs, we investigate how to prevent the functioning of OSNs from being disturbed by adversaries, and we propose SybilDefender, a centralized sybil defense scheme that can effectively detect sybil nodes by analyzing social network topologies. Finally, we study how to protect user privacy on OSNs, and we propose two schemes. MobiShare is a privacy-preserving location-sharing scheme designed for location-based OSNs (LBSNs), which supports sharing locations between both friends and strangers. LBSNSim is a trace-driven LBSN model that can generate synthetic LBSN datasets used in place of real datasets. Combining our work contributes to improving security and privacy in OSNs.

TABLE OF CONTENTS

Acknowledgements vi

Dedication vii

List of Tables ix

List of Figures x

1 Introduction 1

1.1 Improving Infrastructure Reliability of Online Social Networks . . . 5

1.2 Defending Against Sybil Attacks in Online Social Networks 6

1.3 Privacy-Preserving Location Sharing in Location-based Online Social Networks 8

1.4 Modeling Location-based Online Social Networks 9

1.5 Organization 10

2 Related Work 11

2.1 Improving Security of Online Social Networks 11

2.1.1 Improving Infrastructure Reliability of Online Social Networks 12

2.1.2 Defending Against Sybil Attacks in Online Social Networks . 14

2.2 Preserving Privacy in Online Social Networks 16

2.2.1 Privacy-Preserving Location Sharing in Location-based Online Social Networks 16

2.2.2 Modeling Location-based Online Social Networks 19

3 Improving Infrastructure Reliability of Online Social Networks 22
3.1 System Model and Assumptions 26
3.2 Paxos and Mencius revisited 27
3.2.1 Paxos 28
3.2.2 Mencius 29
3.3 Fast Mencius 33
3.3.1 Active Revoke 35
3.3.1.1 Protocol 37
3.3.1.2 Example 39
3.3.2 Multi-instance Propose 40
3.3.2.1 Protocol 42
3.3.2.2 Example 46
3.4 Discussion 48
3.4.1 Correctness of Fast Mencius 48
3.4.2 Dynamically adding and removing replicas 49
3.4.2.1 Adding a replica 50
3.4.2.2 Removing a replica 51
3.4.3 Setting parameters 51
3.5 Evaluation 52
3.5.1 Experimental Setup 53
3.5.2 Evaluation Results 55
3.5.2.1 Commit Latency 55
3.5.2.2 Throughput 58
3.6 Conclusion 59

4 Defending Against Sybil Attacks in Online Social Networks 61

4.1 System Model 64
4.2 SybilDefender Design 66
4.2.1 Sybil Identification Algorithm 67
4.2.1.1 Analysis of the Sybil Identification Algorithm 70
4.2.2 Sybil Community Detection Algorithm 74
4.2.3 Combine Sybil Identification with Sybil Community Detection 78
4.2.4 Limiting the Number of Attack Edges 80
4.2.4.1 Relationship Rating 80
4.2.4.2 Activity Network 82
4.3 Evaluation 82
4.3.1 Data Sets and Experiment Setup 82
4.3.2 Evaluation of the Sybil Identification Algorithm 84
4.3.2.1 Comparison with existing schemes 89
4.3.2.2 Evaluation of the Sybil Identification Algorithm on a Weighted Social Network 92
4.3.2.3 Evaluation of Trust-driven Random Walks 94
4.3.3 Evaluation of the Sybil Community Detection Algorithm . . . 95
4.3.4 Evaluation of the Combo Algorithm 98
4.4 Conclusion 100

5 Privacy-Preserving Location Sharing in Location-based Online Social Networks 101
5.1 System Architecture and Threat Model 104
5.1.1 System Model 104
5.1.2 Trust and Threat Model 106
5.1.3 System Goals 107

5.2 System Design 108
5.2.1 Service Registration 109
5.2.2 Authentication 109
5.2.3 Location Updates 110
5.2.4 Querying Friends' Locations 113
5.2.5 Querying Strangers' Locations 116
5.3 Security Analysis 118
5.4 Evaluation 122
5.4.1 System Implementation 123
5.4.2 Experimental Results 125
5.5 Conclusion 128

6 Modeling Location-based Online Social Networks 130
6.1 Datasets 133
6.2 Data Analysis 134
6.2.1 Number of check-ins 135
6.2.2 Displacement of consecutive check-ins 139
6.2.3 Temporal interval of consecutive check-ins 141
6.2.4 Distance between friends 142
6.2.5 Number of friends 144
6.2.6 Venue popularity 144
6.3 Modeling LBSNs 144
6.3.1 Generating the initial location 145
6.3.2 Building the friendship graph 146
6.3.3 Generating the check-ins 148
6.4 Model Verification 151

6.4.1 Experimental Setup 151
6.4.2 Evaluation Results 151
6.5 Conclusion 156

7 Conclusion 158

References 162

VITA 174

ACKNOWLEDGEMENTS

The work in this dissertation would not have been done without my advisor, Dr. Qun Li. I would like to express my deepest appreciation to him for his guidance.

I am grateful to my committee members, Dr. Weizhen Mao, Dr. Haining Wang, Dr. Gang Zhou, and Dr. Gexin Yu, for their valuable advice.

I would like to thank my parents. Their selfless love and support empower me to advance in my life.

I would also like to thank my dear friends, Fengyuan Xu, Hao Han, Yifan Zhang, Lingfei Wu, Bo Dong, Feng Yan, Han Li, Hui Zhou, Shuai Yang, and Haowei Zhai, for their company and help through the years.

This Ph.D. is dedicated to my parents who are always standing by me along the way...

LIST OF TABLES

3.1 Implementations of three protocols 53

4.1 Notations used in the analysis 70

4.2 10 sybil nodes per attack edge (10000 sybil nodes) false positive and negative rates 85

4.3 5 sybil nodes per attack edge (5000 sybil nodes) false positive and negative rates 86

4.4 1 sybil node per attack edge false positive and negative rates . . 87

4.5 10 sybil nodes per attack edge (100000 sybil nodes) false positive and negative rates 88

4.6 5 sybil nodes per attack edge (50000 sybil nodes) false positive and negative rates 88

4.7 False rates of SybilLimit on the Facebook data set 89

4.8 False rates of the sybil identification algorithm on a weighted social network 92

4.9 False rates of the sybil identification algorithm operating with trust-driven random walks 94

4.10 Performance of the sybil community detection algorithm (1000 attack edges) 96

4.11 Performance of the sybil community detection algorithm (10000 attack edges) 97

4.12 Performance of the sybil community detection algorithm on a weighted social network 97
4.13 Accuracy of the Combo algorithm 98

5.1 Summary of notations 108

6.1 Statistics of the datasets 133
6.2 False positive and negative rates of SybilDefender 155

LIST OF FIGURES

1.1 OSN components and the security and privacy threats 2

3.1 System model 25

3.2 Message pattern of Paxos 27

3.3 Message pattern of Mencius 29

3.4 A system with a slow replica 34

3.5 Instance assignment scheme of Fast Mencius 35

3.6 Message flow of Active Revoke 37

3.7 Multi-instance Propose 44

3.8 Adding a replica 49

3.9 Message flow of adding a replica 49

3.10 Network topology of a three-replica system 54

3.11 Average commit latency of all the replicas 55

3.12 Average commit latency of non-slow replicas 55

3.13 Average commit latency 55

3.14 Average commit latency of non-slow replicas (without *Help* messages) 56

3.15 Peak throughput without any slow replica 57

3.16 Peak throughput with a slow replica 58

3.17 Peak throughput with a slow replica 58

4.1 Pre-processing results and calculated expectation values 71

4.2 Pre-processing results and theoretical approximate results 71
4.3 Our Facebook application: Rate Your Relationships 81
4.4 Difference between the coverage of random walks originating from honest nodes and from sybil nodes 84
4.5 Comparison between the false positive and negative rates of SybilDefender and those of SybilLimit 87
4.6 Comparison between the average running time to test one node by SybilDefender and that by SybilLimit 88

5.1 System architecture 105
5.2 Authentication 110
5.3 Location update 111
5.4 Querying friends' locations 113
5.5 Querying strangers' locations 116
5.6 Client interface 124
5.7 Power consumption of the client 125
5.8 CPU utilization of the cellular tower service 128
5.9 Memory usage of the cellular tower service 128

6.1 Exponentially truncated power law distribution of check-in numbers.135
6.2 Several other fits on the Foursquare check-in number data. . . . 138
6.3 Two-segment distribution of displacements (km). 138
6.4 Several other fits on the Foursquare displacement (km) data. . . 140
6.5 Two-segment distribution of temporal intervals (seconds). 140
6.6 Truncated Weibull distribution of friend distances (km). 142
6.7 Power law distribution of friend numbers. 142
6.8 Power law distribution of venue popularity. 143

6.9 The number of users versus the number of venues in each cell. . 145
6.10 Power-law venue popularity based on the first check-in of each user in the Foursquare dataset. 145
6.11 Ring area to search for the destination node. 147
6.12 Friend distances (km) based on average locations. 152
6.13 Radius of gyration (km). 153
6.14 Inter-checkin time (seconds) of all the check-ins, rescaled by dividing by the average time interval. 153
6.15 Number of friends of each user, rescaled by dividing by the average friend number. 154
6.16 Venue popularity, rescaled by dividing by the average venue popularity. 155

Chapter 1

Introduction

In the past few years, online social networks (OSNs) have gained soaring popularity and are among the most frequently visited sites on the Web [7]. For instance, Facebook alone has around 1.1 billion active users as of March 2013, 60% of which log on Facebook everyday [6]. OSNs provide social connection, communication, and storage applications for users. Through the services provided by OSNs, users establish and strengthen relations with each other by sharing thoughts, activities, photos, locations, and other personal information.

However, the worldwide adoption of OSNs also raises significant security and privacy concerns. First, the proper functioning of large-scale online services like OSNs requires consensus among servers, which may be compromised when some servers fail by crashing for unexpected reasons, such as power outages, physical damage, and online attacks. Second, because of the soaring popularity of OSNs and the vast amount of private and possibly sensitive data they collect, they are the ideal target of various attacks, such as sybil attack [35], fishing attack [55], de-anonymization attack [105], and spam. Through these attacks, attackers profit by compromising the operation of OSNs, polluting

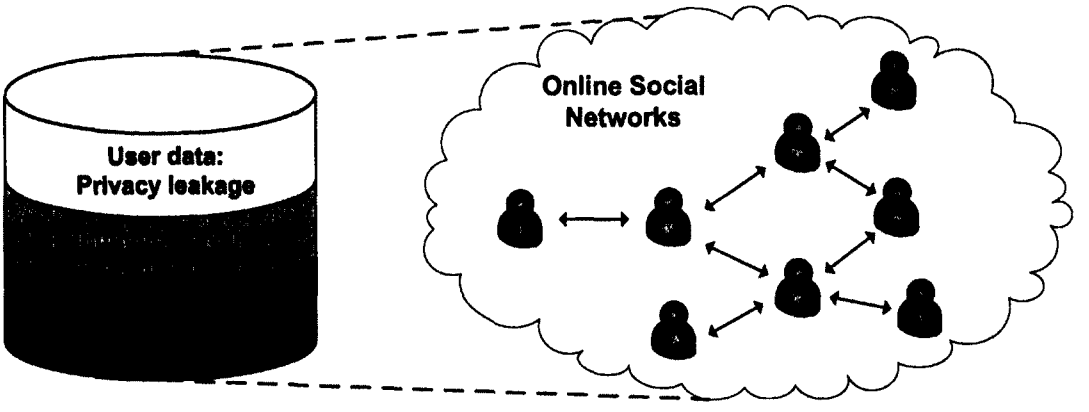


Figure 1.1: OSN components and the security and privacy threats

OSNs with fake information, and cheating legal users. Third, OSNs concentrate data from millions of users under a single administrative domain. This centralization introduces the possibility of large-scale privacy breaches from intentional or unintentional data disclosures. As a result, users' private information may be leaked and abused, such as sexual preferences, political and religious views, phone numbers, occupations, identities of friends, and photos. The privacy threats are more serious with regard to location-based online social networks (LBSNs), a category of OSNs where users share whereabouts with their friends, because on LBSNs users' profile information is correlated with their physical locations.

With OSNs central to so many peoples's lives, it is critical to address the rising tension between the value of participation and the security and privacy threats to OSN users. Users would like to continue using OSNs, but they also want to protect their privacy and avoid becoming attack victims [68]. In this dissertation, we study the security and privacy issues of OSNs following a bottom-up approach. The building of OSNs involves using the underlying servers as the infrastructure, implementing OSN functionalities, and collecting/managing user

data. Starting from the bottom of OSNs, to host millions of users, OSNs are typically running on a large number of servers, which may crash unexpectedly and may be compromised by adversaries. These servers compose the infrastructure of OSNs, whose reliability decides the availability of the OSN services. Built on the infrastructure, the OSN services implement the functionalities provided by OSNs, such as friendship creation, user communication, profile updates, and information sharing. These functionalities are the core value of OSNs, through which OSNs attract millions of users. However, these functionalities also tend to be disturbed by various malicious behaviors on OSNs, in which case the adversaries gain an illegitimate advantage. Finally, during the use of the functionalities provided by OSNs, users have generated a tremendous amount of data, some of which are highly sensitive and private. These data are collected by OSNs, and user privacy will be compromised if their data are leaked and abused. Therefore, in terms of security and privacy, OSNs can be generally divided into three components, the infrastructure layer, the function layer, and the user data, as shown in Figure 1.1. The infrastructure layer includes the servers on which OSNs are deployed, the function layer stands for the functionalities provided by OSNs, and the user data refer to the large amount of data generated by users during their use of OSNs. In this dissertation, to improve security and privacy in OSNs, for each component we analyze and address a representative security/privacy issue. Combining our work contributes to building more secure and privacy-preserving OSNs. The details of our work are presented below.

The infrastructure reliability is critical to the running of OSNs, which are typically running on a large number of servers concurrently. How to reach consensus among those servers is non-trivial, especially considering that those

servers may be compromised by adversaries and fail by crashing. A technique that is widely used to tolerate faults in such an environment is state machine replication, in which a deterministic service is replicated across multiple failure-independent servers. To improve the infrastructure reliability of OSNs, in this dissertation, we analyze the state of the art state machine replication solutions, and propose Fast Mencius, a crash-fault tolerant state machine replication protocol that has high performance in wide-area networks.

A reliable infrastructure is necessary but not sufficient to guarantee the safety of OSNs, as the functioning of OSNs may be disturbed by adversaries who behave maliciously. Adversaries can spam OSNs, cheat legal users, and manipulate social connections. All these malicious behaviors, however, require that the adversaries control a large number of bogus accounts. Because of the lack of verification methods currently on OSNs, it is easy for an adversary to create many fake accounts without being detected. Using these fake accounts the adversary can compromise the operation of OSNs or pollute OSNs with false information. This is called the sybil attack, a fundamental form of attack threatening the security of OSNs. To address this issue we present SybilDefender, a sybil defense mechanism that can effectively detect sybil nodes by analyzing the topology of social graphs.

Our first two work improve the security of the infrastructure layer and the function layer of OSNs, respectively. However, one problem that still waits to be solved is how to protect user privacy on OSNs. OSNs collect a vast amount of private data about users over time, and user privacy may be compromised even when OSNs are functioning appropriately. Among the various types of user information stored by OSNs, the most sensitive is users' physical locations. Once leaked, adversaries can extract users' activities, interests,

habits, and even health conditions from their location traces. By analyzing the functionalities of OSNs we find that the reason why OSNs record users' location information is to provide the location-sharing features, which allow users to share their whereabouts with one another. This, nevertheless, is at the cost of sacrificing user privacy. To address this issue we present MobiShare, a privacy-preserving location-sharing scheme for location-based OSNs (LBSNs). Through this scheme we show that locations can be shared between both friends and strangers on LBSNs without compromising user privacy. Additionally, we also consider protecting user privacy on LBSNs from another perspective. It is known that LBSNs intentionally or unintentionally release their datasets, e.g., for commercial purposes or research purposes, which compose a source of privacy leakage. Even if the datasets are anonymized before being published, user identities can still be recovered from the anonymized location traces and social graphs. To fundamentally solve this problem, we present LBSNSim, a trace-driven model for generating synthetic LBSN datasets that capture the characteristics of the real datasets. The synthetic LBSN datasets can be used as replacements of original datasets, so as to eliminate the privacy risks associated with releasing the real LBSN datasets.

1.1 Improving Infrastructure Reliability of Online Social Networks

The most general approach to implementing a highly reliable service is state machine replication [50], where a deterministic service is replicated across a set of failure-independent servers (replicas), and the replicas consistently change

states by applying commands from an agreed sequence. State machine replication has been extensively explored by previous research [23,64,71] and used in real systems [20,52]. Mencius [71] is the state of the art state machine replication protocol that tolerates crash failures, which has high performance in wide-area networks with its rotating-leader design. However, Mencius has its own weakness. As it is stated by the authors: ``Mencius's commit latency is limited by the slowest replica" [71]. In wide-area networks where link delays are typically large and unpredictable, it is with high probability that some slow replicas exist in the stage machine replication system. With Mencius, the commits of commands at the fast replicas are all delayed by the slowest replica.

To address the weakness of Mencius, we present Fast Mencius, a multi-leader state machine replication protocol. Fast Mencius enhances Mencius with two mechanisms, Active Prepare and Multi-instance Propose. Active Prepare ensures that a fast replica does not need to wait for the slow ones to skip or to propose commands, and Multi-instance Propose gives the slow replicas opportunities to propose their own commands. Our simulation results demonstrate that in presence of slow replicas, the commit latency of Fast Mencius is significantly lower than that of Mencius.

1.2 Defending Against Sybil Attacks in Online Social Networks

OSNs are vulnerable to sybil attacks [35], in which an adversary creates many bogus identities called sybils, and compromises the running of the system or pollutes the system with fake information. Sybil attack is not only a hot research

topic, but also a realistic threat to existing systems. A recent attack to Apple Store shows that sybil attacks can be effectively launched in practice [3].

To prevent the adversary from creating many sybil identities, the previous sybil defense schemes [33, 98, 106, 111, 112] are built upon the assumption that all the relationships in social networks are trusted and they reflect the trust relationships among those users in the real world, and thus an adversary cannot establish many relationships with the honest users. However, it has been shown that this assumption does not hold in OSNs [17]. In addition, the large sizes of OSNs require that any scheme aiming to defend against sybil attacks in OSNs should be efficient and scalable. Some previous schemes are computationally intensive, which cannot scale to networks with millions of nodes [33, 106], while other schemes suffer from high false rates [98, 111, 112].

To address the weaknesses of previous work, we present SybilDefender, a centralized sybil defense mechanism that leverages the network topologies to defend against sybil attacks in OSNs. Based on performing a limited number of random walks within the social graph, SybilDefender is efficient and scalable to large social networks. Our evaluation on two 3,000,000 node real-world social topologies shows that SybilDefender outperforms the state of the art by one to two orders of magnitude in both accuracy and running time. It can effectively identify sybil nodes and detect the sybil community surrounding a sybil node, even when the number of sybil nodes introduced by each attack edge is close to the theoretically detectable lower bound.

1.3 Privacy-Preserving Location Sharing in Location-based Online Social Networks

Compared with traditional OSNs, LBSNs take a step further in that they provide the location-based features, which are a missing link between the real world and OSNs. Instead of letting users explicitly enter their locations, the recent smartphone platforms that support various localization technologies make it much easier for users to access and share their locations with each other.

Although location sharing is a fundamental component of LBSNs, it also raises significant privacy concerns. LBSNs collect a large amount of location information over time, and users' privacy is compromised if their location information is abused by adversaries controlling the LBSNs. Without a guarantee of privacy, users may be hesitant to share locations through LBSNs [14, 68]. To address this issue, we present MobiShare, a privacy management system that supports privacy-preserving location sharing in LBSNs. MobiShare is flexible to support a variety of location-based applications. It enables sharing locations between both trusted social relations and untrusted strangers, and it supports range query and user-defined access control. In MobiShare, neither the social network server nor the location server has complete knowledge of the users' identities and locations. Users' location privacy is protected even if one of the entities colludes with malicious users. Our evaluation results show that MobiShare consumes a very limited amount of system resources on the mobile devices, and it only incurs a small overhead on cellular infrastructure.

1.4 Modeling Location-based Online Social Networks

The soaring adoption of LBSNs makes it possible to analyze human socio-spatial behaviors based on large-scale realistic data, which is important to both the research community and the design of new location-based social applications. Users of LBSNs check-in at different venues (e.g., airports, restaurants) and these check-ins are broadcasted to their friends. In this way users share with friends information about the places they visited. These check-ins, combined with the online friendship connections revealed through the LBSNs, provide an unprecedented opportunity to study human socio-spatial behaviors.

However, performing direct measurements on LBSNs is impractical, because of the security mechanisms of existing LBSNs and the high time and resource costs. The problem is exacerbated by the scarcity of available LBSN datasets, which is mainly due to the privacy concerns. The LBSN datasets reveal users' sensitive private information, such as interests, habits, and health conditions, even if these datasets are anonymized before being published. As a result, only a very few number of LBSN datasets are publicly released. To address this issue, we analyze the characteristics of original LBSN datasets, and present LBSNSim, a trace-driven model for generating synthetic LBSN datasets that capture the properties of the real datasets. Our findings reveal that LBSNs share many universal social and spatial features, and our evaluation demonstrates that the proposed model provides an accurate representation of the target LBSNs

1.5 Organization

This dissertation is organized as follows. In Chapter 2, we present the related work. In Chapter 3, we present Fast Mencius, a crash-fault tolerant state machine replication protocol. In Chapter 4, we present SybilDefender, a security mechanism to defend against sybil attacks in OSNs. In Chapter 5, we present MobiShare, a privacy management system that supports flexible privacy-preserving location sharing in LBSNs. In Chapter 6, we present LBSNSim, a trace-driven LBSN model that can be used to generate synthetic LBSN datasets, and in Chapter 7 we conclude the dissertation.

Chapter 2

Related Work

In this chapter, we present the related work. Following the organization of this dissertation, we first present the work related to the security problems we study in this dissertation, and then we present the work related to the privacy issues we study. A survey of security and privacy issues in OSNs can be found at [82].

2.1 Improving Security of Online Social Networks

For the infrastructure of OSNs, the most general approach to implementing a highly reliable service is state machine replication [50], which has been widely explored by previous research [23, 64, 71] and used in real systems [20, 52]. In this dissertation, we consider the problem of how to build state machine replication systems that can tolerate crash failures, whose related work are presented in Section 2.1.1.

OSNs are venerable to various attacks, such as sybil attack [35], fishing attack [55], de-anonymization attack [105], and spam. Among those attacks sybil attack is the most widely explored by previous research [33, 98, 106, 111,

112]. In this dissertation we propose our own sybil defense scheme that can effectively detect sybil nodes in OSNs, whose related work are presented in Section 2.1.2.

2.1.1 Improving Infrastructure Reliability of Online Social Networks

The well known Paxos protocol [64] was proposed to build replicated state machines that tolerate crash failures in an asynchronous environment. However, its performance is limited by the single leader, which leads to low throughput and high latency. To address this problem, Mao et al. proposed Mencius [71], a multi-leader state machine replication protocol. Derived from Paxos, Mencius is designed to achieve high performance in wide-area networks.

Fast Paxos [67] is an extension of Paxos that admits two execution modes. The classic mode is similar to Paxos, while the fast mode allows clients to directly send their commands to the acceptors. Fast Paxos suffers from collisions, which happens when acceptors receive the commands in different orders. When the number of commands is large, the possibility of collisions is high, and Fast Paxos will have a higher average latency than Paxos, as collisions need to be resolved with a time-consuming recovery procedure. It has also been shown that in wide-area networks, Fast Paxos has a significant probability of having a higher latency than Paxos even in runs without collisions [56]. As for throughput, Fast Paxos cannot outperform Paxos, since the number of messages sent/received by each acceptor is the same with Paxos with no collision, and larger than Paxos when collisions happen.

Several consensus protocols deal with collisions by running Paxos and Fast

Paxos concurrently. The scheme proposed by Charron-Bost and Schiper achieves the minimum latency between Paxos and Fast Paxos only in failure-free runs [25], while Hybrid Paxos [34] and Paxos-MIC [53] have a higher message complexity and thus lower throughput than Paxos. Besides, these protocols rely on a single leader. The unbalanced communication pattern makes them unable to fully utilize the available resources. Yabandeh et al. proposed a protocol for the special case of three replicas [109]. The basic idea is to use a single acceptor and to switch it with another acceptor in the case of failure. The message complexity is thus reduced.

Generalized Paxos [66] reduces collisions by allowing the replicated state machine to commit commands in different but equivalent orders. Similarly, Mencius allows commands x and y to be committed in any order when they are commutable [71], which means committing y after x produces the same system state as committing x after y. This out-of-order commit is used to reduce the extra delay incurred by the delayed commit issue in Mencius.

The authors of Mencius also applied the rotating-leader scheme to Byzantine fault tolerance, and proposed RAM [72], a low latency BFT protocol for wide-area networks. Similarly, EBAWA [99] is a BFT protocol that adopts the rotating-leader design. Different from RAM, EBAWA requires $2f + 1$, instead of $3f + 1$, replicas to tolerate f faulty replicas. It uses A2M [29], a trusted component on the servers to reduce the number of replicas and communication steps required for reaching agreements.

2.1.2 Defending Against Sybil Attacks in Online Social Networks

The initial paper by Douceur [35] on sybil attacks shows that sybil attacks cannot be prevented unless special assumptions are made. Bazzi and Konjevod [15] assume that an attacker has only one network position, and all the sybil nodes created by the attacker have similar network coordinates [79]. However, this defense will be broken once an attacker possesses multiple network positions.

One promising way to defend against sybil attacks in social networks is to leverage the social network topologies. Yu et al. proposed decentralized algorithms, SybilGuard [112] and SybilLimit [111], to determine whether a suspect node is sybil or not. SybilGuard and SybilLimit both rely on the assumption that social networks are fast mixing (explained later), and the number of attack edges is limited. To identify sybil nodes, the schemes make use of random routes, a special kind of random walks in which each node uses a pre-computed random permutation as a one-to-one mapping from incoming edges to outgoing edges. SybilGurad suffers from high false negatives, as each attack edge may introduce $O(\sqrt{n}\log n)$ sybil nodes without being detected. The improved version of SybilGuard, SybilLimit, reduces this value to $O(\log n)$, which is still larger than the proved lower bound $\Omega(1)$ [111] by a $\log n$ factor. Moreover, to detect the sybil region with SybilGuard or SybilLimit, all the suspect nodes in the social graph need to be tested. By contrast, with our sybil community detection algorithm, the sybil community around a sybil node can be detected in one run of the algorithm.

GateKeeper [98] is another decentralized sybil defense scheme that heavily relies on the assumption that the social networks are random expander. This is

a strong assumption which has not been validated by previous research. Our evaluation shows that GateKeeper suffers from high false positive and negative rates and cannot effectively identify sybil nodes on the real-world asymmetric social topologies.

SybilInfer [33], a centralized sybil defense algorithm, leverages a Bayesian inference approach that assigns a sybil probability, indicating the degree of certainty, to each node in the network. It achieves low false negatives at the cost of high computation overhead. The overall time complexity of SybilInfer is $O(|V|^2 \log |V|)$, where V is the set of vertices in the social graph. In the evaluation SybilInfer handled networks with up to 30K nodes, which is much smaller than the size of regular OSNs. Xu et al. [106] leveraged the edge-betweenness based clustering algorithm proposed by Girvan and Newman [42] to detect sybil areas. The algorithm calculates the shortest path between every pair of nodes within the network in each round, which makes it impractical for even small-sized social networks. In contrast, SybilDefender only relies on performing a limited number of random walks in the social graph, and it is scalable to large networks.

Some previous solutions mitigate sybil attacks through the use of computational puzzles or CAPTCHAs [83, 87]. These approaches can limit the rate at which sybil identities are introduced into the systems, but they cannot identify the existing sybil identities. They can be used in conjunction with the scheme proposed in this chapter. Besides this, sybil attacks can also be mitigated by limiting social information sharing in physical proximity [97, 110].

2.2 Preserving Privacy in Online Social Networks

The privacy issues of OSNs have also been extensively studied by previous research, such as the design of privacy-preserving OSNs [13], privacy risks in OSNs [21, 60], privacy-enhancing approaches for OSNs [38, 48], and users' privacy preferences on OSNs [96]. In this dissertation we focus on protecting users' location privacy [94], and we propose two scheme. MobiShare is a privacy management scheme that supports privacy-preserving location-Sharing on LBSNs, whose related work are presented in Section 2.2.1. LBSNSim is a trace-driven model for generating synthetic LBSN datasets that can be used as replacements for original LBSN datasets, whose related work are presented in Section 2.2.2.

2.2.1 Privacy-Preserving Location Sharing in Location-based Online Social Networks

Techniques to preserve location privacy. The most intuitive way to protect location privacy is to replace the user identity in the location data with an untraceable ID, i.e., a pseudonym [16]. However, previous research has shown algorithms that can break pseudonyms and reconstruct tracks based on the anonymized location traces from multiple users [46, 51, 63]. These algorithms use multi-target tracking and k-means clustering to re-identify individuals from the anonymized traces. Therefore, simply replacing user identities with pseudonyms is not enough to guarantee location privacy.

To address this issue a number of stronger techniques have been proposed. Gruteser and Grunwald [45] introduced k-anonymity for location privacy. When

a user requests a location-based service, the proposed scheme computes a cloaking region that contains the requesting user and at least $k-1$ other users, and uses this region as the user's location to request the service. Along this direction a series of work have been done. For instance, Gedik and Liu [41] proposed letting users specify their own k values and minimizing the size of cloaking regions, while Xu and Cai's work [107, 108] considers leveraging the historic location samples and allows a user to express his privacy requirement by specifying a public region, which the user feels comfortable if the region is reported as his location. The weakness of these cloaking schemes is that they undermine the accuracy of the responses from the service provider. This defeats the location-based applications in which users would like to learn the precise location of one another. Also, it has been shown that even with cloaking user privacy may still be compromised [43].

Kido et al. [58] proposed protecting location privacy through dummy location updates. The basic idea is to append multiple dummy location updates to each real location update sent to the service provider. The authors discussed the issues including realistic dummy movements and reduction of communication costs. Compared with the cloaking mechanisms, the advantage of using dummy location updates is that the service accuracy is not hurt. In our work we leverage both pseudonyms and dummy location updates to protect users' location privacy.

Privacy-preserving location-based applications. SmokeScreen [32] is a privacy management system designed to enable privacy-preserving presence sharing in location-based social services. The users share their presence with each other by broadcasting short-range wireless messages. Similarly, SMILE [70] is a location-based social service in which trust among users

is established solely based on the shared encounters, so called ``missed connections". Co-located peers use a wireless link to generate a symmetric key at each encounter. Later on they use this key to identify each other and protect their communication. Popa et al. proposed PrivStats, a privacy-preserving system for computing aggregate statistics over location data without sacrificing accountability [84]. Location privacy and accountability in PrivStats are achieved through an aggregate statistics protocol and a zero-knowledge proof of knowledge protocol, respectively. Chen et al. considered a similar problem [26]. The authors proposed a scheme for statistical queries over distributed user data that provides distributed differential privacy. An honest-but-curious proxy is designed to add noise to query answers such that the querier cannot detect the presence or absence of a single user or a set of users. Narayanan et al. presented several protocols that support privacy-preserving proximity testing [77]. In their design the problem of private proximity testing is reduced to the underlying cryptographic problem of private equality testing (PET), which is considered in settings with or without the involvement of a server. In addition, Koi, a privacy-preserving location-based matching platform, was proposed in [47]. Koi allows applications to specify a location event of interest and notifies the application when there is a location match. Privacy is achieved through a design comprising two non-colluding entities that together implement location matching.

As for privacy-preserving location sharing, Puttaswamy and Zhao proposed an approach [85] in which a user's location update is first encrypted with a session key shared with all his friends, and then stored on an untrusted third-party server. In this way only the friends of the user can access and decrypt his location data. This scheme, however, is inflexible in that it restricts location sharing to existing relations. Moreover, users have no control over how their locations

are shared with others, i.e., all the friends of a user can access his location information no matter where he is. Zhong et al. proposed three protocols [115], which only support learning friends' whereabouts, and each protocol has its own drawbacks. For example, the Louis and the Pierre protocols only allow a user to learn the distances between himself and his friends. Additionally, all three protocols are computationally intensive and do not apply to mobile devices.

2.2.2 Modeling Location-based Online Social Networks

Previous studies have attempted to investigate the check-in properties of LBSNs. Cheng et al. studied human mobility patterns revealed by check-ins, and explored factors that influence mobility [27]. They found that LBSN users follow the ``Levy Flight" mobility pattern, which is characterized by a mixture of short, random movements with occasional long jumps. Similar findings have been observed in previous research based on cellphone call data [44], bank note dispersal [18], and GPS traces [86]. Scellato et al. presented a study of the socio-spatial properties of LBSNs [90]. They found that about 40% of all pairs of friends are within 100 km, and that in LBSNs long range social ties have a higher probability of occurrence than in other social systems. Brown et al. proposed a method to extract place-focused communities from the social graph of LBSNs by annotating the edges with check-in information [19]. The method can extract group of users connected not only by social ties, but also by the common places they visited. User behavior with regard to LBSNs has been analyzed by Lindqvist et al. [68]. The authors conducted interviews and surveys to investigate how and why people use LBSNs, and their privacy concerns related to the location-sharing functions.

Researchers have also utilized the social and geographic information of LBSNs for location prediction and friendship prediction. Backstrom et al. observed that the likelihood of friendship drops monotonically as a function of distance, and exploited this relation to infer Facebook users' registered addresses based on their friends' addresses [12]. Cho et al. proposed a location prediction model built on the idea that human check-ins are based on the movement between two latent states ``work" and ``home". They also considered the influence of social network structure on individuals' mobility [28]. To recommend new venues to LBSN users, Noulas et al. proposed a recommendation scheme relying on performing personalized random walks on a user-place network, where a user is linked to her friends and the venues she has visited before [80]. As for friendship prediction, Scellato et al. built a supervised learning framework that exploits the features extracted from LBSNs to predict new links between friends-of-friends and place-friends, which are the users visiting the same place [91].

Sala et al. explored the feasibility of replacing real social graphs of online social networks with synthetic graphs generated from calibrated graph models [88]. The authors compared six existing graph models. They found that two models consistently generate synthetic graphs with common graph metric values similar to those of the original graphs, and one produces high fidelity results in application-level tests. In a followup work the authors investigated how to share social network graphs without compromising user privacy [89]. Kim and Leskovec presented a multiplicative social-attribute graph model modeling the structure of social networks where nodes have attribute information [59]. The model considers nodes with categorical attributes and the probability of an edge between a pair of nodes as the product of individual attribute link formation affinities. Pfeiffer et al. proposed a random graph generation model that

combines the standard Chung-Lu model with edges that are formed through transitive closure [54]. The new model's expected degree distribution is equal to the degree distribution of the original input social graph.

Chapter 3

Improving Infrastructure Reliability of Online Social Networks

Starting from the infrastructure layer of OSNs, we first consider how to improve infrastructure reliability of OSNs. As the reliance of our society on wide-area computing services such as OSNs grows, tolerating faults in these services is increasingly important. State machine replication [50] is the most general approach to implementing a highly reliable service. With this approach, a deterministic service is replicated across a set of failure-independent replicas, and the replicas consistently change their states by applying commands from an agreed sequence. Each command in the sequence is chosen by a consensus instance. State machine replication has been widely explored by previous research [23, 64, 71] and used in real systems [20, 52]. In this chapter we consider the problem of building replicated state machines in wide-area networks that tolerate crash failures.

It has been proved that under pure asynchronous circumstances, no consensus protocol can ensure both safety and liveness, even when a single replica

can fail [39]. However, the impossibility result can be circumvented by making weak assumptions about the synchrony of the system. The well-known Paxos protocol [64] was proposed to build replicated state machines that tolerate crash failures in an asynchronous environment, assuming the existence of an eventual leader election mechanism. The simplicity of Paxos enables it to achieve good throughput during normal execution. However, its performance is limited by the single leader. With Paxos, all the client commands need to be first forwarded to the leader. In a wide-area network where a client is not co-located with the leader, this incurs high transmission delay. Moreover, the leader in Paxos is responsible for proposing all the client commands. This one-to-many communication pattern leads to the fact that when the system is network-bound, the throughput of the system is limited by the bandwidth of the links incident upon the leader, and the available bandwidth of other links is not fully used. Also, since the leader needs to process much more messages than the other replicas, when the system is CPU-bound, the throughput of the system is limited by the leader's processing power. The variants of Paxos, like Fast Paxos [67], Generalized Paxos [66], and Hybrid Paxos [34], reduce Paxos's latency when certain conditions are met, but they still suffer from the weakness that the leader is a bottleneck of the system.

To address this problem, Mao et al. proposed Mencius [71], a multi-leader state machine replication protocol. Derived from Paxos, Mencius is designed to achieve high performance in wide-area networks. The basic idea of Mencius is to partition the sequence of consensus instances among all the replicas. By doing so, each replica proposes the received client commands using its allotted instances, and the replicas in total can service more commands, compared with the single-leader schemes. With Mencius, the network resources are more

fully used, and the load of being the leader is amortized among all the replicas. Besides, clients can use their local replica to propose commands, and thus the latency for clients to receive replies is reduced.

Mencius, however, has its own weakness. As it is stated by the authors: ``Mencius's commit latency is limited by the slowest server'' [71], where commit latency is defined as the interval between when a replica proposes a command and when the command is committed by the replica. State machine replication requires that a replica commit a command only when it learns the command has been chosen in a consensus instance, and it has learned and committed the commands chosen in all the previous instances. With Mencius, to commit a command chosen in instance i, a replica has to wait for all the other replicas to skip, by proposing *no-op*, or to propose commands in their allotted instances smaller than i. In wide-area networks where link delays are typically large and unpredictable, it is with high probability that some slow replicas, whose transmission delay is larger than the others, exist in the system. With Mencius, the commits of commands at the fast replicas are all delayed by the slowest replica.

This chapter presents Fast Mencius, a multi-leader state machine replication protocol that is derived from Mencius. With Fast Mencius, the commit latency of fast replicas is not limited by the slowest replica, while the slow replicas can still have their proposed commands chosen by the replicated state machine. To achieve this, Fast Mencius enhances Mencius with two mechanisms, Active Prepare and Multi-instance Propose. In presence of slow replicas, Active Prepare ensures that a fast replica does not need to wait for the slow ones to skip or to propose commands, and Multi-instance Propose gives the slow replicas opportunities to propose their own commands. We also investigate issues including how to dynamically add and remove replicas in state machine repli-

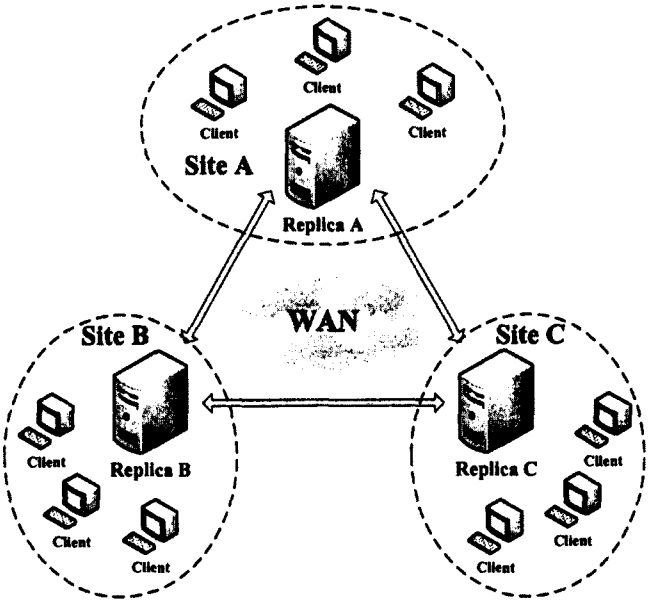


Figure 3.1: System model

cation systems using Fast Mencius or Mencius. Evaluation results show that with Fast Mencius, the commit latency of the non-slow replicas is independent of the delay of the links connecting the slowest replica, and the throughput of Fast Mencius is significantly larger than that of Paxos even in presence of slow replicas.

The rest of this chapter is organized as follows: Section 3.1 presents the system model and assumptions. Section 3.2 revisits Paxos and Mencius, from which Fast Mencius is derived. The design of Fast Mencius is described in detail in Section 3.3. Section 3.4 presents discussions related to our protocol. Section 3.5 evaluates Fast Mencius. In Section 3.6 we conclude the chapter.

3.1 System Model and Assumptions

Like Mencius, we model a system as n sites interconnected by a wide-area network. The link delays of the wide-area network are large and may have high variance. Each site contains a state machine replica and a group of clients, which communicate through links with high bandwidth and small delay. The n replicas communicate to implement a crash fault-tolerant replicated state machine. Replicas can fail by crashing, and may later recover. They have access to stable storage to record their states, which will be used after they recover from a failure. The system is asynchronous, with no bound on message transmission delay. As a crash fault-tolerant protocol, Fast Mencius requires $2f + 1$ replicas to handle up to f concurrent faulty replicas, with a quorum size of $f+1$. Figure 3.1 shows an example system of 3 replicas.

To use the service, clients send commands to their local replica. The replicas establish a total order for all the commands by running a sequence of consensus instances. Each command is chosen in one instance. Replicas commit the commands in the decided order. Once a command is committed, the reply is sent from the local replica back to the client generating the command.

The safety requirements of consensus are as follows:

- *Nontriviality*: Any chosen command must have been proposed.
- *Stability*: a replica can learn at most one command in a consensus instance.
- *Consistency*: Two replicas cannot learn different commands in a consensus instance.

The liveness requirement of consensus is as follows:

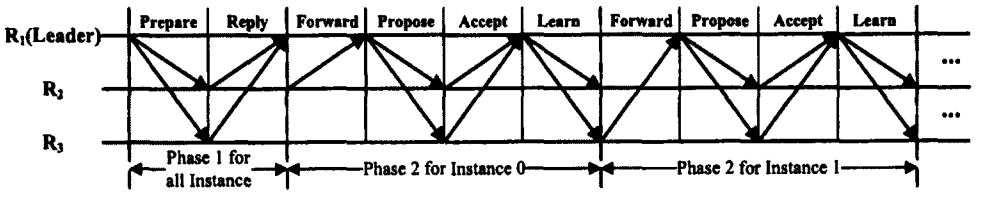


Figure 3.2: Message pattern of Paxos

- *Termination*: Every non-faulty replica eventually learns some command in a consensus instance.

Fast Mencius shares the same assumption with Mencius that the communication channels are FIFO, and messages between two correct replicas are eventually delivered, but there is no upper-bound on message delivery time. In practice, the communication channels can be implemented by UDP with retransmission and flow control or TCP. To circumvent the FLP impossibility result, which states that no consensus protocol can ensure both safety and liveness even when a single replica can fail [39], we assume that each replica has access to a failure detector $\Diamond\mathbb{P}$, which guarantees eventually all faulty servers and only faulty servers are suspected [24]. Like Mencius, the failure detector is only used by Fast Mencius to guarantee liveness. Fast Mencius is safe even if the failure detector makes an unlimited number of mistakes.

3.2 Paxos and Mencius revisited

Before we describe how Fast Mencius is derived from Mencius, we first briefly review Paxos and Mencius.

3.2.1 Paxos

Paxos was first proposed in [64], and was further explained in [65]. Each Paxos instance consists of one or more rounds, and each round is divided into two phases. Paxos assumes Ω, a failure detector that provides eventual unique leader election functionality. The safety of Paxos is guaranteed even if multiple leaders are elected, but a unique leader is required to ensure liveness.

Phase 1, also known as the Prepare Phase, is only run when there is a leader change, e.g., the previous leader has crashed. A new leader l starts a higher numbered round r by sending $Prepare(r)$ to all the replicas. If r is greater than any other round replica a has heard of, a sends $Reply(r, vrnd, vval)$ back to l, where $vrnd$ is the highest-numbered round in which a has accepted a command, and $vval$ is the command a accepted in round $vrnd$. l collects the $Reply$ messages from a majority of replicas (including the one from itself). If no $Reply$ indicates a previously accepted command, then l is free to propose any command in Phase 2. Otherwise, l picks up $vval$ in the $Reply$ messages with the highest value of $vrnd$, and proposes this command in Phase 2. Note that Phase 1 can be run concurrently for an unlimited number of future instances, and thus its cost is amortized.

Upon receiving a command from a client, a non-leader replica forwards the command to l. To propose a command, l starts Phase 2, also known as the Propose Phase, by sending $Propose(r, val)$ to all the replicas, where val is the command. If replica a has not heard of a round numbered greater than r, it accepts the proposal and sends $Accept(r, val)$ to l. After l collects the $Accept$ messages from a majority of replicas, it learns that val has been chosen in this consensus instance, and sends $Learn(val)$ to all the replicas. Upon receiving

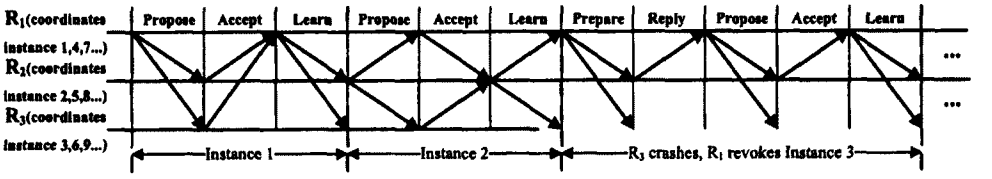


Figure 3.3: Message pattern of Mencius

the *Learn* message, a learns that the outcome of this instance is val. a commits val if it has learned and committed the commands chosen in all the previous instances. The client generating the command gets the execution result from its local replica. In Phase 2, The *Learn* messages can be omitted by letting each replica send its *Accept* message to all the replicas, instead of only to the leader. In this case, a replica learns that a command has been chosen once it receives *Accept* messages from a majority of replicas. This option, however, decreases learning latency at the cost of increased message complexity, which in turn reduces the system throughput.

By analyzing the protocol, it is easy to see that the leader is a bottleneck in Paxos. With Paxos, the leader needs to send, receive, and process many more messages than the other replicas. As a result, the throughput of the system is limited by the available bandwidth of the network links incident upon the leader when the system is network-bound, as well as the prossing power of the leader when the system is CPU-bound. Figure 3.2 shows the message pattern of a sequence of Paxos instances. Note that instances may overlap in practice.

3.2.2 Mencius

As a multi-leader state machine replication protocol, Mencius [71] is derived directly from Paxos. To order the commands, Mencius runs an unbounded se-

quence of simple consensus instances. Let *no-op* be a command that leaves the replica state unchanged and generates no response. In each simple consensus instance, only one replica, which is called the coordinator, can propose any command, while the other replicas can only propose *no-op*. Simple consensus allows a coordinator to skip its instance with only one communication step: other replicas learn that *no-op* has been chosen in this instance once they receive a $Skip$ message, which is a $Propose$ message that proposes *no-op*, from the coordinator. The sequence of simple consensus instances is partitioned among all the replicas. Each replica is the coordinator (default leader) of an unbounded number of instances. The simplest way to assign instances to replicas is in a round-robin fashion: the i^{th} replica coordinates instance $cn+i$, where $c \in \{0, 1, 2, ...\}$ and n is the number of replicas.

In Mencius, simple consensus is implemented by Coordinated Paxos [71]. Coordinated Paxos differs from Paxos in that each Coordinated Paxos instance starts from the state in which the coordinator had run the Prepare Phase of Paxos for some initial round r. This is safe because in each instance all the replicas agree that the coordinator is the default leader.

Mencius is built upon the following rules.

Rule 1: Each replica p maintains I_p, the sequence number of the next available simple consensus instance it coordinates. I_p is also called the *index* of p. Upon receiving a command from a local client, p proposes the command in instance I_p with round r, and updates I_p to the next instance it coordinates. This rule ensures that a replica proposes a command immediately after receiving it from a client.

Rule 2: If replica p receives a $Propose$ message in instance i and $i > I_p$, before accepting the proposal, p updates I_p such that the new index $I'_p = min\{i' :$

$p\,coordinates\,instance\,i' \wedge i' > i\}$. p also executes skip actions, by proposing *no-op*, for each of the instances in range $[I_p, I'_p)$ it coordinates. With this rule, the replicas proposing commands less frequently, e.g., fewer clients in their sites generate commands, skip their instances.

Rule 3: By accessing the failure detector $\Diamond\mathbb{P}$, if replica p suspects replica q has failed, and C_q is the smallest instance that is coordinated by q and not learned by p, then p revokes all the instances in range $[C_q, I_p]$ that q coordinates. Note that revocation is done by running both the Prepare Phase and Propose Phase of Paxos, as shown in Figure 3.3. With this rule, a crashed replica cannot prevent other replicas from committing their learned commands.

Rule 4: If replica p proposes a command $v \neq$ *no-op* in instance i, and p learns that *no-op* is eventually chosen in this instance, which means p has been falsely suspected and instance i has been revoked by another replica, then p proposes v again in a higher instance it coordinates. This rule allows a falsely suspected replica to propose a command multiple times, until false suspicions eventually cease.

These four rules guarantee that any client command sent to a correct replica is eventually committed, and it takes the minimal two communication steps ($Propose$ and $Accept$) for a proposing replica to learn the outcome of a consensus instance. However, the message complexity depends on the rates at which the replicas propose client commands: when the replicas propose commands at different rates, they need to execute many skip actions, by sending $Skip$ messages. Mencius solves this problem with two optimizations.

Optimization 1: If replica p receives a $Propose$ message from q in instance i and $i > I_p$, p uses the $Accept$ message that replies the $Propose$ to promise q that p will not propose any client command in instances smaller than i in the

future, i.e., *no-op* has been chosen in these instances. In this case, p does not need to send $Skip$ messages to q. p does not send $Skip$ messages to other replicas either. Instead, for each of these replicas, p waits for a future $Propose$ message sent to that replica, to promise not to propose any client command in smaller instances. This optimization is valid since Mencius assumes FIFO channel: before a replica receives an $Accept$ or $Propose$ message from p, it must have received all the previous $Propose$ messages from p, so it can safely infer the instances p skips.

Optimization 2: A replica p propagates $Skip$ messages to another replica q if the total number of delayed $Skip$ messages to q is larger than some constant α, or the messages have been deferred for more than some time t. This rule is used to limit the delay of the propagation of $Skip$ messages due to Optimization 1.

Figure 3.3 shows the message flow of Mencius. Compared with Paxos, Mencius has a more balanced communication pattern, which better utilizes the available bandwidth and achieves higher throughput when the system is network-bound. The load of being the leader is distributed among all the replicas, which eliminates the computational bottleneck in Paxos and enables higher throughput when the system is CPU-bound. With Mencius, each replica proposes a command immediately after receiving it from a client, and learns the command is chosen once it receives Accepts from a majority of replicas. This eliminates the learning latency incurred by the $Forward$ and $Learn$ messages in Paxos.

3.3 Fast Mencius

In wide-area networks where the link delays are influenced by the network traffic and are highly unpredictable, it is with high probability that the connection speed of a replica drops below the other replicas, and becomes a slow one. Mencius has high performance in wide-area networks. However, its commit latency is limited by the slowest replica. We explain this through an example. Considering the scenario in Figure 3.4, the system consists of 3 replicas, among which A is a slow replica: the links connecting A have a much larger delay. Assume at time 0 each replica proposes its first command: A proposes in instance 1; B proposes in instance 2; C proposes in instance 3. With Mencius, at 50 ms B and C receive the $Propose$ message from each other, and reply with an $Accept$ message. At 100 ms B and C have collected $Accept$s from a majority of replicas (including the one from themselves), and learn that their proposed command has been chosen. However, at this time, B cannot commit its command chosen in instance 2, because it does not know the outcome of instance 1. Nor can C commit its command chosen in instance 3. At 150 ms, B and C receive the $Learn$ message from each other. B learns the outcome of instance 3, and C learns the outcome of instance 2. They still cannot commit their proposed command, as the result of instance 1 is unknown. At 500 ms, B and C receive the $Propose$ from A, and reply with $Accept$. At 1000 ms, A learns its command is chosen in instance 1, and broadcasts the $Learn$ message. A also commits this command. At 1500 ms, B and C receive the $Learn$ message from A and learn the outcome of instance 1. Only at this time can B and C commit their proposed command. As a result, although it only takes B and C 100 ms to learn their command has been chosen, their commit latency is 1500 ms, three

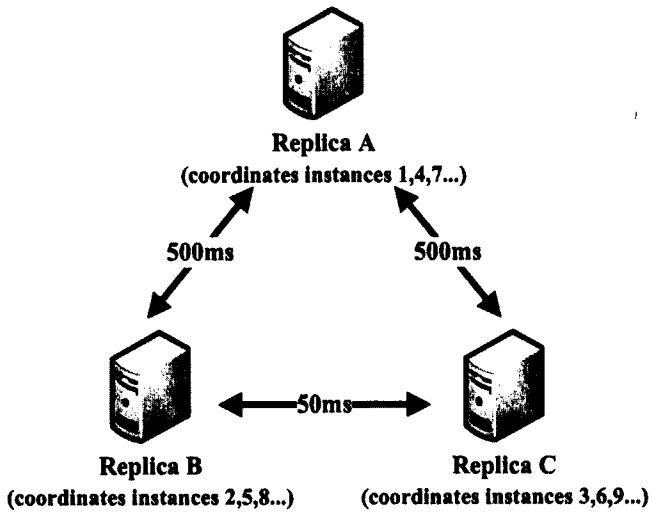


Figure 3.4: A system with a slow replica

one-way delays of the links incident upon A. Ironically, the slow replica A has a smaller commit latency of 1000 ms.

State machine replication requires that a replica commit a command only when it learns the command has been chosen in a consensus instance, and it has learned and committed the commands chosen in all the previous instances. With Mencius, to commit a command v chosen in instance i, a replica p has to wait for all the other replicas to propose commands, or to skip, by proposing *no-op*, in their allotted instances smaller than i. Otherwise, there will be gaps in the command sequence learned by p, and p cannot commit v.

Fast Mencius addresses this problem with two mechanisms, Active Revoke and Multi-instance Propose. With Active Revoke, fast replicas can proceed without being delayed by the slowest replica, while Multi-instance Propose enables slow replicas to have their proposed commands chosen by the replicated state machine, even in presence of Active Revoke and false failure suspicions.

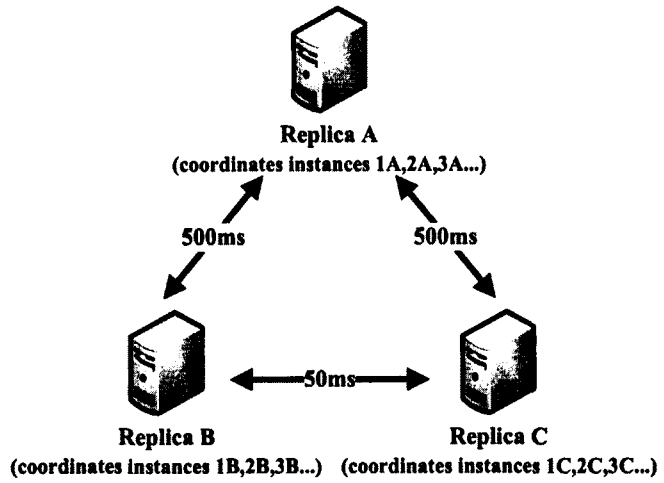


Figure 3.5: Instance assignment scheme of Fast Mencius

3.3.1 Active Revoke

The intuition behind this mechanism is that, instead of requiring fast replicas wait for the messages, such as $Propose$, $Skip$, or $Learn$, from slow replicas to commit their commands, we allow them to actively revoke the instances coordinated by slow replicas, as long as their commit of client commands has been delayed for a sufficiently long time.

Different from Mencius, an instance number in Fast Mencius is of the form $counter \parallel id$, where id is the identifer of the coordinator of this instance. The instance numbers are ordered primarily by $counter$. For the instance numbers with the same $counter$, they are ordered lexicographically by id. For example, the system in Figure 3.5 has three replicas A, B, and C. Then the instance numbers will be ordered as $1A < 1B < 1C < 2A < 2B < 2C$.... This assignment scheme enables dynamically adding and removing replicas, which will be explained in the next section. Besides, given one instance number, all the replicas know unambiguously who is the coordinator of this instance.

Active Revoke is triggered if Condition 1 is met:

Condition 1 The commit of one of replica p's proposed commands has been deferred by an unlearned instance for more than some time τ, and the failure detector indicates the coordinator of the unlearned instance is alive.

If this condition is met, we say p is delayed by the coordinator of the unlearned instance. To expedite its advancement, p will revoke the instances coordinated by the slow replica that prevent it from committing its commands, by running both the Prepare Phase and Propose Phase of Paxos. Note that in Mencius, revocation is done only when a replica suspects another has failed. In Fast Mencius, revocation is also used to help the fast replicas speed up their commits. The problem arises from this design is that if multiple replicas concurrently revoke the instances coordinated by a slow replica, there will be competing $Prepare$ messages: a $Prepare$ with a higher round number will suppress a $Prepare$ with a lower round number. As a result, only the sender of the $Prepare$ with the highest round number can finish the revocation process, and the work done by other replicas is wasted. In fact, when multiple replicas are delayed by a slow replica, after one of them finishes revocation, the others can learn the outcome of the revocation with the broadcasted $Learn$ message. Therefore, the number of concurrently revoking replicas should be limited to save resources. This is achieved through the use of $Help$ messages in our design.

$Help$ messages are small status-checkers the replicas use to inquire other replicas about a particular piece of missing information, and determine the appropriateness of sending the following $Prepare$ message. After a replica p initiates Active Revoke, it first sends a $Help$ message to all the other replicas, which contains the instance numbers of the consensus instances it wants to revoke. Upon receiving the $Help$ message, another replica q replies with an Ack message, which includes the following information.

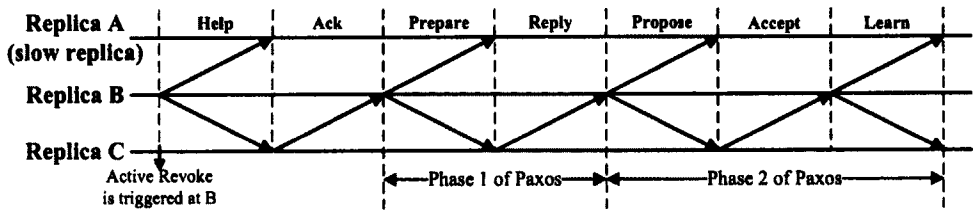


Figure 3.6: Message flow of Active Revoke

1. If q has learned the outcomes of some of the queried instances, then Ack includes the learned commands.

2. If q is revoking some of the queried instances, which means it has sent the $Prepare$ message in these instances, then Ack includes these instance numbers.

p collects the Acks from $\lfloor \frac{n-1}{2} \rfloor$ other replicas. Through these replies, p learns the commands chosen in some queried instances directly if they are in the Acks. p also knows which queried instances are currently being revoked by other replicas, and p will wait for these replicas to finish revocation and broadcast the results. Note that since p only waits for the Acks from a minority of replicas, it is rational for p to assume that the senders of these Acks are the relatively fast ones, so it can rely on them to do revocation. p revokes only the queried instances whose results are still unknown and no Ack indicates they are currently being revoked.

3.3.1.1 Protocol

The full Active Revoke mechanism is as follows.

Help. If Condition 1 is met, then p initiates Active Revoke. Assume the coordinator of the unlearned instance is q. Let C_q be the smallest instance coordinated by q and not learned by p, and let I_p be p's index, i.e., the next available

instance coordinated by p. p sends $Help(C_q, C'_q)$ to the other replicas, where $C'_q = max\{i : q\ coordinates\ instance\ i \wedge i < I_p\}$.

Upon receiving the $Help$ message, another replica r composes a reply as *Ack(instance entries)*. The message may include multiple instance entries for the instances within the range $[C_q, C'_q]$ that q coordinates. If r has learned the results of some of these instances, then the corresponding entry is $\langle i, `l', v\rangle$, where $i$ is the instance number, and $v$ is the learned command. Else if $r$ is revoking some of these instances, then the corresponding entry is $\langle i, `r'\rangle$. Otherwise, Ack is an empty message.

Revoke. p waits for the Acks from $\lfloor \frac{n-1}{2} \rfloor$ other replicas. Assuming set S consists of the sequence numbers of the queried instances whose results are still unknown and no Ack indicates they are currently being revoked, p starts revoking the instances in S by running both the Prepare Phase and Propose Phase of Paxos, as shown in Figure 3.6, with an optimization. The optimization is, if a replica already accepts a command that is not *no-op* in an instance, and it starts revoking this instance, then it broadcasts a special $Prepare$ message. Another replica's $Reply$ to this $Prepare$ does not convey the command it has accepted in this instance, if any. This optimization reduces the overhead of revocation in Fast Mencius. It is valid because by the definition of simple consensus, only the coordinator can propose any command in an instance, and the other replicas can only propose *no-op*. Therefore, if any replica accepts a command that is not *no-op* in an instance, it must have been proposed by the coordinator, and there is no need to let the $Reply$ contain the command if the revoking replica already knows about it.

During Active Revoke, if p learns the results of all the queried instances, either from q or from other replicas, then p stops doing Active Revoke immedi-

ately. Also, p performs Active Revoke for each unlearned instance coordinated by q only once.

3.3.1.2 Example

We use an example to illustrate how Active Revoke expedites the advancement of the fast replicas. Consider the scenario in Figure 3.5. Assume at time 0 each replica proposes its first command: A proposes in instance *1A*; B proposes in instance *1B*; C proposes in instance *1C*. At 100 ms B and C learn that their proposed command has been chosen. However, they cannot commit, because they don't known the result of instance $1A$. Let the timeout threshold τ be 100 ms. Then at 200 ms both B and C initiate Active Revoke by broadcasting the $Help$ message. At 300 ms B and C receive the Ack from each other. Since neither of them knows the outcome of instance $1A$ or is currently revoking $1A$, both B and C decide to revoke $1A$ and broadcast the $Prepare$ message. Note that with Paxos, the round numbers are partitioned among the replicas, e.g., in a round-robin fashion, so two replicas will never send $Prepare$ messages with the same round number. Assuming the $Prepare$ sent by C has a higher round number than the one sent by B, C's $Prepare$ will suppress B's $Prepare$, i.e., B will respond to C's $Prepare$, while C will not respond to B's $Prepare$. At 400 ms C receives the $Reply$ from B. Since neither B nor C has accepted a command in instance $1A$, C proposes *no-op* in instance $1A$. At 500 ms C receives the $Accept$ from B, and learns that *no-op* is chosen in instance $1A$. C sends the $Learn$ message to all the replicas and commits the command it proposes in instance $1C$. At 550 ms B receives the $Learn$ message from C, and learns the result of instance $1A$, so B can commit the command it proposes in instance

$1B$. As a result, Active Revoke reduces the commit latency of B and C from 1500 ms to 550 ms and 500 ms, respectively. This improvement is achieved because Active Revoke enables the fast replicas to advance without learning from the slow replicas, and thus their commit latency only depends on the delay of the links connecting themselves, instead of the delay of the links incident upon the slowest replica.

In this simplified example, B and C concurrently revoke the instances coordinated by A. This is because here B and C propose client commands at the same rate, and the link delay between A and B is identical to that between A and C. As a result, B and C initiate Active Revoke simultaneously. In real environments where replicas propose commands at varying rates and the network conditions are heterogeneous, it is with high probability that replicas enter the Active Revoke state at different time, in which case the $Help$ messages will limit the number of concurrently revoking replicas. Notice that even when more than one replica revokes the instances coordinated by a slow replica, our protocol does not have the liveness problem faced by Paxos when it has multiple leaders, since a replica is allowed to do Active Revoke for each unlearned instance coordinated by a slow replica only once.

3.3.2 Multi-instance Propose

Active Revoke speeds up the advancement of the fast replicas, by allowing the fast replicas to revoke the instances coordinated by the slow replicas. In the worst case, however, a slow replica may never be able to commit its proposed commands. We explain this using the scenario in Figure 3.5.

Assuming at time 0 each replica proposes its first command, replica A pro-

poses a command v. With Active Revoke, replica B and C broadcast their $Prepare$ message at 300 ms. When A's $Propose$ message is delivered at B and C at 500 ms, they will not accept this proposal, because they have responded to a $Prepare$ with a higher round number. As a result, in instance *1A*, A cannot collect $Accept$s from a majority of replicas. At 1000 ms, A receives the $Learn$ message from C, which informs that *no-op* has been chosen in instance $1A$. Then A knows that instance $1A$ has been revoked by C. Following Rule 4 of Mencius, A proposes v again in the next available instance $2A$. However, assuming at this time B or C also proposes a new command, the commit of this command will be delayed by A, which triggers Active Revoke again and results in instance $2A$ being revoked. In the worst case, this situation may repeat an unbounded number of times, and A can never commit v.

Besides Active Revoke, the instances coordinated by slow replicas may also be revoked because of false failure suspicions. Since we assume the failure detector is unreliable, it may make mistakes. Compared with fast replicas, the slow replicas are more likely to be falsely suspected of having failed. When false suspicions happen, following Rule 3 of Mencius, the suspecting replicas will revoke the instances coordinated by the suspected slow replicas, which prevents the commands proposed by the slow replicas from being chosen and committed. Mencius's assumption on the failure detector is that false suspicions will eventually cease. However, this may take an unbounded length of time. Therefore, the slow replicas will suffer from large and unpredictable commit latency if they are falsely suspected.

To ensure the progress of Fast Mencius, i.e., any client command sent to a correct replica is eventually committed, and also to limit the commit latency of the slow replicas in presence of false suspicions, the slow replicas should

be given opportunities to have their proposals chosen by the replicated state machine, even when other replicas initiate Active Revoke or false suspicions happen. This is achieved through our Multi-instance Propose mechanism. The intuition behind this mechanism is that instead of letting a slow replica, whose instances are being revoked by others, propose its command in one instance at a time, we let it propose its command in multiple instances simultaneously. In this way, even if other replicas revoke some of these instances, the slow replica can still commit its command if it is chosen in at least one of these instances.

3.3.2.1 Protocol

Multi-instance Propose is triggered if the following condition is met:

Condition 2 None of γ consecutive commands proposed by replica p is chosen by the replicated state machine.

A replica learns that its proposed command is not chosen, by receiving a $Learn$ message from another replica that informs *no-op* has been chosen in the instance in which it proposed the command. If replica p finds that none of its γ consecutively proposed commands is chosen, it can deduce that it is slow relative to others, and other replicas are revoking its instances. Under this circumstance, p initiates Multi-instance Propose to ensure that its commands can be successfully committed.

MultiPropose. With Multi-instance Propose, p proposes a command v in a block of consensus instances simultaneously, by broadcasting $MultiPropose(I_1, I_2, r, v)$. This message means that p proposes v in all the instances within the range $[I_1, I_2]$ that it coordinates. Since p is the default leader of these instances, the round number r is 0. To determine I_1 and I_2, let C_p be the smallest instance

among the γ revoked instances coordinated by p, and I_p be p's index. Then $I_1 = I_p$, and the number of instances coordinated by p within the rage $[I_1, I_2]$ is equal to the number of instances coordinated by p within the range $[C_p, I_p)$. For example, if $C_p = 7p$, and $I_p = 10p$, then $I_1 = 10p$, and $I_2 = 12p$. After sending out the $MultiPropose$ message, p updates I_p to the next available instance it coordinates, which is $13p$ in this case. Following Optimization 1 of Mencius, p also uses this $MultiPropose$ message to promise not to propose any client command in instances smaller than I_1 in the future. That is, another replica learns that the outcomes of all the instances smaller than I_1 and coordinated by p, in which it has not received any proposal from p, are *no-op*, immediately after it receives the $MultiPropose$ message.

MultiAccept. After another replica q receives the $MultiPropose$ message from p, if its index I_q is smaller than I_1, q updates I_q such that its new index $I'_q = min\{i : q \text{ coordinates } i \wedge i > I_1\}$. This is the application of Rule 2 of Mencius in our Multi-instance context. q also checks in which instances within the range $[I_1, I_2]$ that p coordinates it can accept v, i.e., it has not responded to a $Prepare$ message with a higher round number in these instances. If q finds that it can accept v in at least one instance coordinated by p within the range $[I_1, I_2]$, then q composes a $MultiAccept$ message, which includes the sequence numbers of all the instances in which it can accept v, and sends the message back to p. Following Optimization 1 of Mencius, q also uses this $MultiAccept$ to promise p that it will not propose any client command in instances smaller than I_1 in the future.

Learn. p waits for the $MultiAccept$s from a majority of replicas, and broadcasts $Learn(i, v)$, where i is the smallest instance that appears in all the *MultiAccept*s from a majority of replicas. Note that there is at least one instance

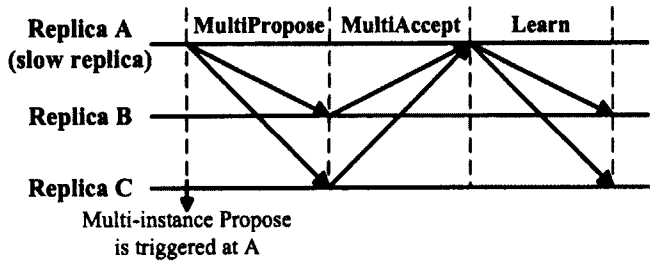


Figure 3.7: Multi-instance Propose

that appears in all the $MultiAccept$s received by p, which is instance I_2. Here is an example. Let the number of replicas be 5, with a quorum size of 3. Assume $I_1 = 10p$, $I_2 = 12p$, and the first 3 $MultiAccept$s received by p are $MultiAccept(10p, 11p, 12p)$, $MultiAccept(11p, 12p)$, $MultiAccept(12p)$. Then p broadcasts $Learn(12p, v)$. This $Learn$ message does not mean that v will be definitely committed at instance $12p$, because the results of instances $10p$ and $11p$ are still unknown. After sending out $Learn(12p, v)$, if p receives $MultiAccept(11p, 12p)$ from another replica, then it broadcasts $Learn(11p, v)$. Otherwise, if p receives two $MultiAccept(10p, 11p, 12p)$ messages, then it broadcasts $Learn(10p, v)$.

Commit. All the replicas will commit v at the smallest possible instance. This means that a replica can make the decision to commit v at instance i, only after it learns *no-op* has been chosen in all the instances smaller than i in the instance block p uses to propose v. In this case the replica will commit *no-op* at all the other instances in the instance block. Following the previous example, a replica decides to commit v at instance $12p$ only after it receives $Learn(10p,$ *no-op*$)$, $Learn(11p,$ *no-op*$)$, and $Learn(12p, v)$; it decides to commit v at instance $11p$ only after it receives $Learn(10p,$ *no-op*$)$, and $Learn(11p, v)$; it decides to commit v at instance $10p$ immediately after it receives $Learn(10p, v)$. Note that the $Learn$ messages that choose *no-op* come from the replicas who

are revoking instances coordinated by p. The message flow of Multi-instance Propose is shown in Figure 3.7.

It is possible that all the instances in the instance block p uses to propose v are revoked by other replicas. If p receives the $Learn$ messages from other replicas that indicate *no-op* has been chosen in all these instances, then p doubles the size of the instance block and proposes v again. By increasing the size of the instance block exponentially, p can quickly get an opportunity to have its proposal chosen.

A replica initiates Multi-instance Propose once for only one command, i.e., p proposes v using Multi-instance Propose, but it proposes the commands after v still following Mencius, as long as Multi-instance Propose is not triggered again. The reasoning is that after successfully proposing a command with Multi-instance Propose, a slow replica catches up with the advancement of the other replicas, and others' revocations will not prevent the commands proposed by the slow replica from being chosen, assuming the slow replica's connection speed is not dropping even more. Therefore, Multi-instance Propose is designed to provide a one-time boost to the slow replica, and it will go back to the normal mode of operation afterwards. When a replica is in the Multi-instance Propose state, it is not allowed to start Active Revoke. This is rational because being in the Multi-instance Propose state means this replica is relatively slow, and it should not revoke the instances coordinated by others. The replica is allowed to initiate Active Revoke only after it decides at which instance to commit the command proposed with its $MultiPropose$ message, i.e., when it quits the Multi-instance Propose state.

After a replica q receives the $MultiPropose$ message from p, and before it decides at which instance to commit v, it does not update its index following

Rule 2 of Mencius when it receives subsequent $Propose$s from p. Optimization 1 of Mencius does not apply here either: q does not use the $Accept$ that replies a subsequent $Propose$ from p to promise not to propose any client command in a smaller instance. To let p know that this $Accept$ has a different semantics, q adds a flag in the message body. This design gives p priority to propose v: the other replicas need to revoke all the instances in the instance block step by step such as to invalidate the $MultiPropose$ message from p.

3.3.2.2 Example

We use an example to explain how Multi-instance Propose enables a slow replica to propose its commands. Consider the system in Figure 3.5. Assuming each replica proposes a command every 10 ms, replicas A, B, and C propose their 1st command at time 0, the 2nd command at 10 ms, and so on. In this scenario the consensus instances execute concurrently.

At 100 ms B and C learn that their 1st command has been chosen in instance $1B$ and $1C$, respectively. However, they cannot commit because the result of instance $1A$ is unknown. Assume the threshold τ used to trigger Active Revoke is 100 ms, and the threshold γ to trigger Multi-instance Propose is 10. Then at 200 ms B and C initiate Active Revoke by broadcasting $Help(1A, 22A)$. At 500 ms C finishes revocation and broadcasts the $Learn$ message for instances $1A$, ..., $22A$, informing that *no-op* has been chosen in these instances. The $Learn$ message is delivered at A at 1000 ms, which triggers Multi-instance Propose at A, since more than 10 consecutive commands proposed by A are not chosen. As the smallest revoked instance is $1A$, and A's current index is $102A$, A sets the size of the instance block to 101 and broadcasts $MultiPropose(102A, 202A,$

0, v) to propose a command v at 1010 ms. The $MultiPropose$ message is delivered at B and C at 1510 ms. At this time B and C have revoked all the instances coordinated by A within the range $[1A, 132A]$, following Active Revoke. Therefore, B and C accept v in all the instances coordinated by A within the range $[133A, 202A]$, and their $MultiAccept$s are delivered at A at 2010 ms, when A learns that v has been chosen in these instances. A receives the $Learn$ message from C at 1880 ms that indicates *no-op* has been chosen in all the instances it coordinates within the range $[89A, 110A]$, and the $Learn$ message from C at 2100 ms that informs *no-op* has been chosen in all the instances it coordinates within the range $[111A, 132A]$. Then A can commit v at instance $133A$. All the other instances in the instance block A uses to propose v are considered *no-op*. As a result, the commit latency for v is 1090 ms, slightly larger than a round-trip delay between A and other replicas.

The commands A proposes after v will not be influenced by other replicas' revocations. Following the previous example, C learns it should commit v at instance $133A$ when it finishes revoking the instances coordinated by A within the range $[133A, 154A]$ at 1820 ms, and B learns the same result when it receives the $Learn$ message from C at 1870 ms. After that, when they receive a new $Propose$ message from A, they will follow Rule 2 of Mencius to make their indices match the instance number of the $Propose$, and their future revocations will not prevent the commands proposed by A from being chosen, as long as the network conditions stay the same. This is because when they send out $Prepare$ messages to revoke some of A's instances, they already received the proposals from A for these instances. As a result, they will propose A's proposals in these instances.

3.4 Discussion

3.4.1 Correctness of Fast Mencius

To revoke the instances of slow replicas, Active Revoke uses the Prepare and Propose phases of Paxos, whose correctness has already been proved [64]. The $Help$ messages are status-checkers that determine when revocation should be triggered, and they do not interfere with the revocation mechanism in each consensus instance. Besides, a replica is allowed to perform Active Revoke for each unlearned instance at most once. Thus, we avoid the liveness problem faced by Paxos when it has multiple revoking replicas, in which different replicas keep issuing revocation with increasing round numbers.

The correctness of Multi-instance Propose is guaranteed by ensuring that all the unfailed replicas will make the same decision at which instance to commit the command proposed by Multi-instance Propose. Fast Mencius runs an unbounded sequence of simple consensus instances. Each instance is implemented by Coordinated Paxos [71], which is the same with Paxos except for a different starting state, since in each simple consensus instance the coordinator is the default leader. Paxos ensures that in each consensus instance all the correct replicas learn the same result, so for each instance in the instance block used by a slow replica to propose a command v with Multi-instance Propose, all the correct replicas receive the same $Learn$ message. Also, a replica cannot commit a command unless it learns the commands chosen in all the previous instances. Therefore, all the unfailed replicas will make the same decision at which instance to commit v.

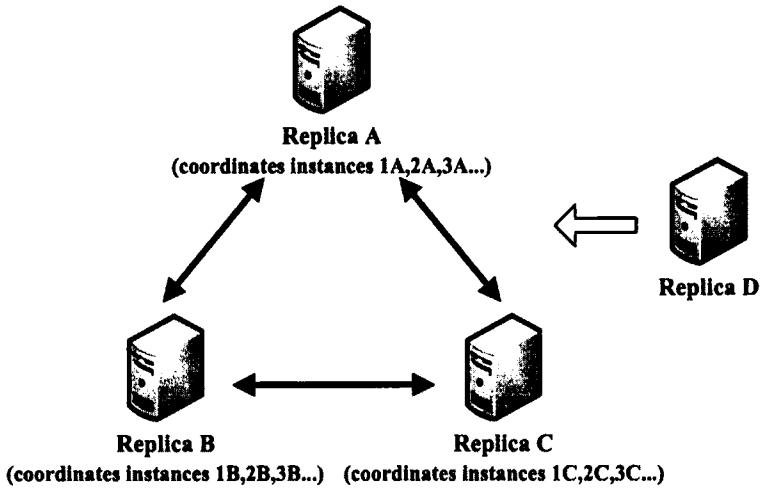


Figure 3.8: Adding a replica

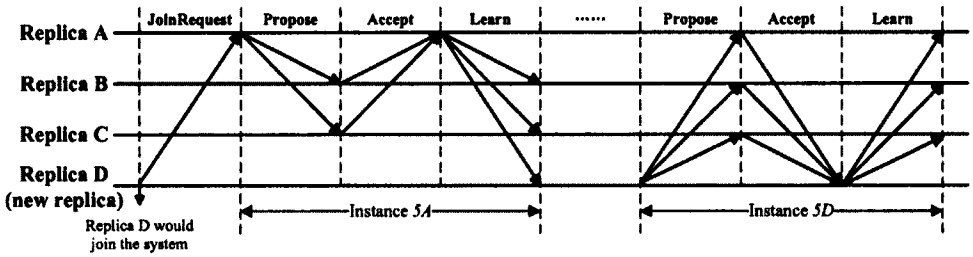


Figure 3.9: Message flow of adding a replica

3.4.2 Dynamically adding and removing replicas

With Mencius or Fast Mencius, each replica is the default leader of an unbounded number of instances. Also, adding and removing replicas involve changing the quorum size in the scale of the whole system. These make adding and removing replicas in Mencius or Fast Mencius without stalling the system and without any extra synchrony assumption a non-trivial problem. In this subsection we discuss how we solve this problem.

3.4.2.1 Adding a replica

As mentioned in Section 3.3.1, each instance number in Fast Mencius is of the form $counter \parallel id$. We assume that this strategy is used to assign consensus instances to replicas. Consider the system in Figure 3.8. Assuming a new replica D will join the system, D first asks one of the replicas already in the system to propose its entrance via a normal $Propose$ message. For example, A proposes this request in instance $5A$. After collecting the $Accept$s from a quorum, which is 2 in this case, of replicas, A sends the $Learn$ message to all the replicas, including D, which informs D's entrance has been chosen in instance $5A$. Upon receiving the $Learn$ message, each replica already in the system learns that D joins the system starting from instance $5B$, and D's coordinated instances, $5D$, $6D$, ..., have been inserted into the sequence of consensus instances executed by the replicated state machine. The replica changes the quorum size from 2 to 3 for each unlearned instance larger than $5A$, where ``unlearned" means it has not received a $Learn$ message of that instance. Also, for each instance larger than $5A$ in which it has sent out a $Propose$ or a $Prepare$, but has not received enough replies from a previous quorum, which is 2, of replicas, the replica sends another copy of the ongoing message to D, and D's reply will be counted to meet the new quorum requirement.

After D sends the entrance request to A, it buffers all the protocol messages from other replicas. Once it receives the $Learn$ message from A that indicates its entrance has been chosen in instance $5A$, D learns that it should participate in the system beginning from instance $5B$. The first instance it uses to propose a command is $5D$. It also replies to all the buffered and future incoming messages following the protocol, which are all for instances larger than $5A$. The

reason why D needs to buffer the messages is that it is possible other replicas' messages are delivered at D before A's $Learn$ message. Figure 3.9 shows the message flow of adding replica D.

3.4.2.2 Removing a replica

Removing a replica is more straightforward. When a replica p is to leave a system, it sends a special message $Quit(i)$ to all the other replicas, where i is its current index, i.e., the next available instance coordinated by p. This message is simultaneously a promise that p will fully participate in all the unfinished instances smaller than i, as well as a notice that it will not be involved in any instance larger than or equal to i. Upon receiving the $Quit$ message from p, all the other replicas can safely assume that prior to instance i, the quorum size stays the same, and the quorum size is recalculated for all the instances thereafter, with one less replica in the system. Also, the replicas learn that starting from instance i, the instances coordinated by p, whose numbers are of the form $counter \parallel p$, are all removed from the sequence of consensus instances executed by the replicated state machine. For the unfinished instances smaller than i, the replicas still send the protocol messages to p, while they will not send any message to p for instances larger than or equal to i. p keeps processing the protocol messages for instances smaller than i, until it has learned the outcomes of all these instances, at which time p can be safely removed from the system.

3.4.3 Setting parameters

Our Active Revoke and Multi-instance Propose mechanisms are triggered by two conditions using parameters τ and γ, respectively. Here we discuss how to

set them.

τ determines when a replica delayed by a slow one should start Active Revoke. In Mencius, the maximum extra latency caused by delayed commit, which happens when there are concurrent $Propose$s, is a round trip delay [71]. Therefore, the minimal value of τ is a round trip delay between the non-slow replicas. A replica can estimate the round trip delay between itself and other non-slow replicas by measuring the time interval between when it sends out a $Propose$ and when it collects $Accepts$ from a majority of replicas. Assuming this delay is d, τ can be set to d. Using a larger τ reduces the number of times Active Revoke is triggered, while it increases the commit latency. In the evaluation we set τ to a round trip delay between the non-slow replicas.

γ determines when a slow replica, whose coordinated instances are revoked by others, should start Multi-instance Propose. Since being revoked in a sequence of consecutive proposed instances happens only when a replica is slow relative to others, γ can be set to the number of commands a replica proposes within a short time period. For example, in our evaluation we set γ to the number of commands a replica proposes within 100 ms.

3.5 Evaluation

In this section we first describe our protocol implementations and experimental setup, and then we present the evaluation results.

Table 3.1: Implementations of three protocols

Protocol	Overlog Rules	C++ LOC
Paxos	53	871
Mencius	59	1037
Fast Mencius	99	2552

3.5.1 Experimental Setup

We implemented Fast Mencius using BFTSim [95], a simulation framework for state machine replication protocols. BFTSim couples a high-level protocol specification language and execution system based on P2 [69] with a network simulator built upon ns-2. The declarative networking language used by P2, OverLog, allows protocol designers to capture the salient points of each protocol, without diving into the details of particular thread packages, messaging modules, and so on. Besides, ns-2's network simulation function enables designers to explore the performance of their protocols under various network conditions that typical testbeds cannot easily address. BFTSim was originally proposed to implement and compare the Byzantine fault-tolerant (BFT) protocols, including PBFT [23], Q/U [9], and Zyzzyva [61], and it faithfully predicts the performance of the native protocol implementations [95]. Note that the functions needed to implement BFT protocols are a superset of the functions needed to implement the crash fault-tolerant consensus protocols, since the latter do not use the crypto operations required by the former. Therefore, BFTSim can also be used to implement and compare the crash fault-tolerant protocols.

For comparison, we also implemented Paxos and Mencius with BFTSim. Each implementation consists of OverLog rules and C++ code. The details of the three implementations are shown in Table 3.1. To further validate the fidelity of BFTSim, we measured the throughput and latency of our Paxos and

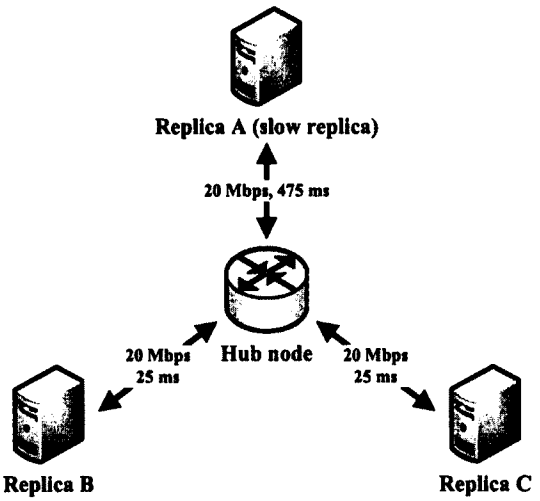


Figure 3.10: Network topology of a three-replica system

Mencius implementations under a range of network conditions, and reproduced the published results. For example, evaluation in the original paper shows that with a three-replica clique topology, when the payload size of each command is 4000 bytes and the total bandwidth is 99 Mbps, the peak throughput of Mencius is 1550 operations per sec (ops), and the peak throughput of Paxos is 540 ops [71]. With our implementations and under the same network condition, the peak throughput of Mencius and Paxos is 1550 ops and 550 ops, respectively. The results demonstrate that BFTSim is accurate enough to be used in evaluating the performance of the consensus protocols we study.

We simulated a star network topology, where the replicas are connected to each other via a hub node. Each link between a replica and the hub node is a duplex link, with a bandwidth of 20 Mbps. The one-way delay between a non-slow replica and the hub node is 25 ms. This gives an RTT of 100 ms between any pair of non-slow replicas. The link between the slow replica and the hub node has a much larger delay. A three-replica topology is shown in Figure 3.10.

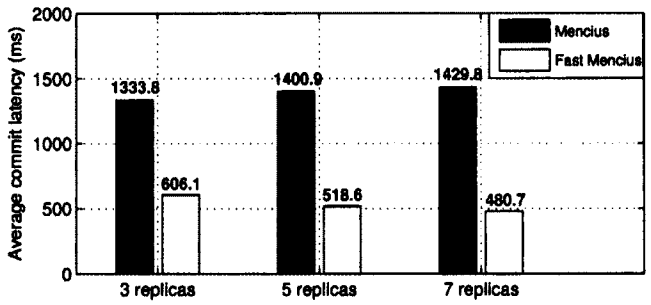


Figure 3.11: Average commit latency of all the replicas

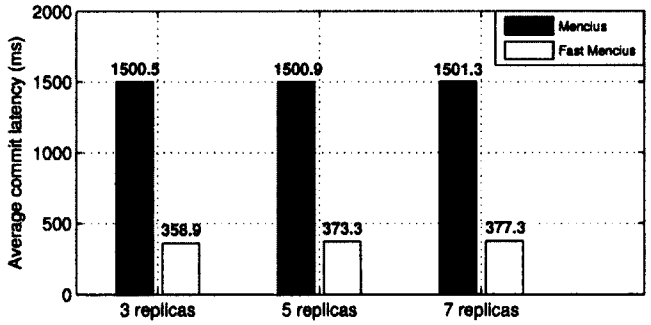


Figure 3.12: Average commit latency of non-slow replicas

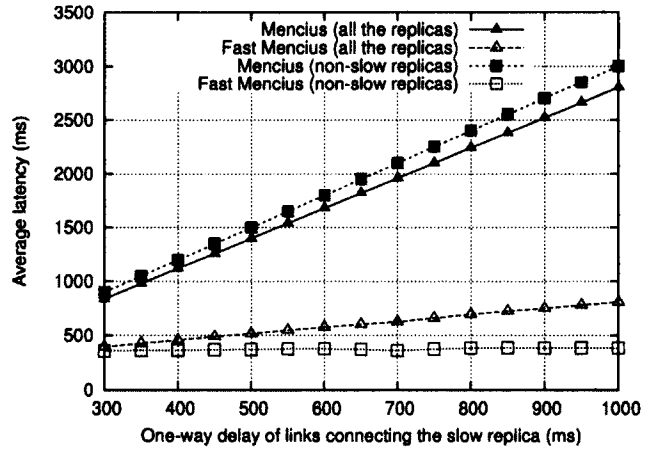


Figure 3.13: Average commit latency

3.5.2 Evaluation Results

3.5.2.1 Commit Latency

As stated in the original paper: ``Mencius's commit latency is limited by the s-lowest server." To get this result, we set the one-way delay between the slow

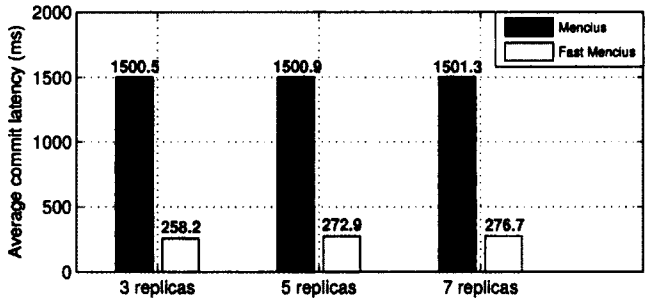


Figure 3.14: Average commit latency of non-slow replicas (without *Help* messages)

replica and the hub node to 475 ms, which gives an RTT of 1000 ms between the slow replica and any non-slow replica. Each replica proposes commands at a rate of 100 ops. The size of payload in each command is 5 bytes. By varying the number of replicas, we got the average commit latency of Mencius. For comparison, we set $\tau = 100\ ms$, $\gamma = 10$, and got the average commit latency of Fast Mencius, as shown in Figure 3.11. The results illustrate that the commit latency of Fast Mencius is significantly smaller than that of Mencius. With the increase of the number of replicas, the average commit latency of Mencius increases, while the average commit latency of Fast Mencius decreases. The reason is shown in Figure 3.12. With Mencius, the commit latency of the non-slow replicas is about 3 times of the delay of the links connecting the slow replica, while the commit latency of the slow replica is 2 times of this delay. With Fast Mencius, the non-slow replicas have a much lower commit latency, and the commit latency of the slow replica is still about 2 times of the delay of the links connecting itself.

We got the commit latency of Mencius and Fast Mencius by varying the delay of the links connecting the slow replica, from 300 ms to 1000 ms with an interval of 50 ms. The system consists of 5 replicas. Figure 3.13 shows the average

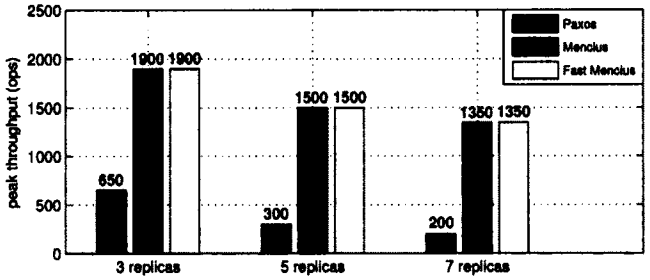


Figure 3.15: Peak throughput without any slow replica

commit latency of all the replicas and that of the non-slow replicas. The results illustrate that when we raise the delay of the links connecting the slow replica, the commit latency of Mencius increases considerably faster than Fast Mencius. Moreover, with Fast Mencius, the commit latency of the non-slow replicas is not influenced by the slow replica. We also measured the commit latency when a non-slow replica fails. The commit latency of the remaining replicas is not influenced. This is because both Active Revoke and Multi-instance Propose only require the participation of $f + 1$ replicas.

Within Active Revoke of Fast Mencius, $Help$ messages are status-checkers used to limit the number of concurrently revoking replicas. Figure 3.14 shows the average commit latency of non-slow replicas without the use of $Help$ messages, in which case a replica directly revokes the instances coordinated by a slow replica through broadcasting $Prepare$ messages, once Condition 1 is met. It is shown that without $Help$ messages, the commit latency of non-slow replicas is further reduced. However, this design increases the number of concurrently revoking replicas, and thus decreases the peak throughput of Fast Mencius.

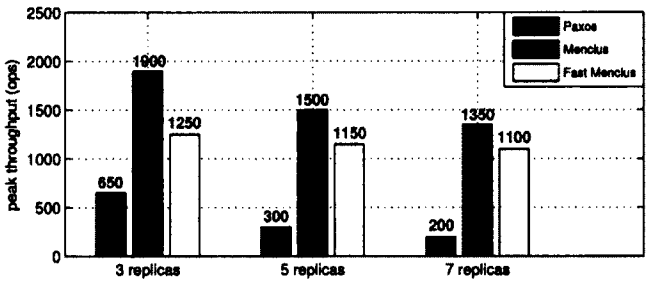


Figure 3.16:

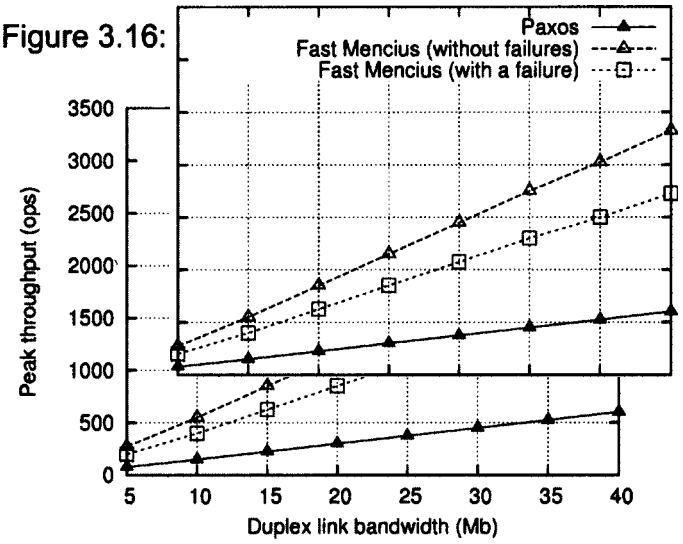


Figure 3.17: Peak throughput with a slow replica

3.5.2.2 Throughput

When there is no slow replica in the system, Fast Mencius behaves the same as Mencius. Figure 3.15 compares the peak throughput of Paxos, which has high throughput due to its simplicity, with that of Fast Mencius, when the delay between every replica and the hub node is 25 ms, and the bandwidth of each duplex link is 20 Mb. The size of payload in each command is 2000 bytes. The results show that, without any slow replica, the peak throughput of Fast Mencius is about n times of that of Paxos, where n is the number of replicas. The reason is that Fast Mencius, derived from Mencius, utilizes available bandwidth in a more balanced way with the rotating-leader design, while the single leader in

Paxos limits the peak throughput it can achieve.

Figure 3.16 compares the peak throughput of Fast Mencius with that of Paxos and Mencius, when there is a slow replica. The delay between the slow replica and any non-slow replica is 500 ms. The results show that Fast Mencius still outperforms Paxos significantly. Also, the throughput difference between Fast Mencius and Mencius, which is the overhead incurred by our mechanisms, becomes smaller when the number of replicas increases. This is because the task of revoking the instances coordinated by the slow replica is done by more replicas.

Figure 3.17 compares the peak throughput of Fast Mencius with that of Paxos, when there is a slow replica, by varying the bandwidth of links from 5 Mbps to 40 Mbps with an interval of 5 Mbps. The delay between the slow replica and any non-slow replica is 500 ms. The system consists of 5 replicas. As expected, the peak throughput of both protocols scales with available bandwidth, while the peak throughput of Fast Mencius increases much faster than Paxos. Figure 3.17 also shows the peak throughput of Fast Mencius when a non-slow replica fails. Compared with the non-failure case, the throughput of Fast Mencius drops by around 25%. The reason is that in a 5-replica system with a slow one, a failure decreases the available bandwidth of the non-slow replicas by 25%.

3.6 Conclusion

In this chapter, we present Fast Mencius, a protocol for state machine replication that tolerates crash failures. Fast Mencius is derived from Mencius, and it enhances Mencius with Active Revoke and Multi-instance Propose. The e-

valuation shows that in presence of slow replicas, the commit latency of Fast Mencius is significantly smaller than that of Mencius.

Chapter 4

Defending Against Sybil Attacks in Online Social Networks

For the function layer of OSNs, we study how to prevent the functioning of OSNs from being disturbed by sybil attacks [35]. It is well known that online services are vulnerable to sybil attacks, in which an adversary creates many bogus identities, called sybil identities, and compromises the running of the system or pollutes the system with fake information. The sybil identities can ``suppress" the honest identities in a variety of tasks, including online content ranking, DHT routing, file sharing, reputation systems, and Byzantine failure defenses.

Sybil attacks can be mitigated by assuming the existence of a trusted authority, which can rate-limit the introduction of fake identities by requiring the users to provide some credentials, like social security number, or by requiring payment. However, such requirements will prevent users from accepting these systems, as they impose additional burdens on users.

Recently, there has been an increasing interest in defending against sybil attacks in social networks [33, 98, 106, 111, 112]. In a social network, two user

identities share a link if a relationship is established between them. Each identity is represented as a node in the social graph. To prevent the adversary from creating many sybil identities, all the previous sybil defense schemes are built upon the assumption that the number of links between the sybil nodes and the honest nodes, also known as *attack edges*, is limited. As a result, although an adversary can create many sybil nodes and link them in an arbitrary way, there will be a *small cut* between the honest region and the sybil region. The small cut consists of all the attack edges and its removal disconnects the sybil nodes from the rest of the graph, which is leveraged by previous schemes to identify the sybil nodes. Note that the solution to this problem is non-trivial, because finding small cuts in a graph is an NP hard problem. To limit the number of attack edges, previous schemes assume that all the relationships in social networks are trusted and they reflect the trust relationships among those users in the real world, and thus an adversary cannot establish many relationships with the honest users. However, it has been shown that this assumption does not hold in some real-world social networks [17].

In the past few years, online social networks (OSNs) have gained great popularity and are among the most frequently visited sites on the Web [7]. The large sizes of these networks require that any scheme aiming to defend against sybil attacks in OSNs should be efficient and scalable. Some previous schemes can achieve good performance on a very small network sample (2000 nodes in [106] and 30000 nodes in [33]), but their algorithms are computationally intensive and cannot scale to networks with millions of nodes. For the schemes that performed evaluation on million-node samples of OSNs, SybilGuard [112] admits $O(\sqrt{n}\log n)$ sybil nodes per attack edge, where n is the number of honest nodes; SybilLimit [111] improves over SybilGuard by accepting $O(\log n)$ sybil

nodes per attack edge, but it is still away from the theoretical lower bound by a $\log n$ factor. Besides, both SybilGuard and SybilLimit identify one sybil node at a time, and thus to detect the sybil region all the nodes in the social graph need to be examined.

To address the weaknesses of previous work, in this chapter we propose SybilDefender, a centralized sybil defense mechanism. It consists of a sybil identification algorithm to identify the sybil nodes, a sybil community detection algorithm to detect the sybil community surrounding a sybil node, and two approaches to limiting the number of attack edges in OSNs. Our scheme is based on the observation that a sybil node must go through a small cut in the social graph to reach the honest region. An honest node, on the contrary, is not restricted. Now if we start from a sybil node to do random walks, the random walks tend to stay within the sybil region. The main contributions of this work include:

- Based on performing a limited number of random walks within the social graphs, our proposed sybil identification and sybil community detection algorithms are more efficient than previous techniques for large social networks.

- We evaluate SybilDefender using two large-scale social network samples from Orkut and Facebook, respectively. The results show that the performance of our sybil identification algorithm approaches the theoretical bound, and it outperforms SybilLimit, the state of the art sybil defense mechanism that applies to large social networks, by one to two orders of magnitude in both accuracy and running time. In addition, our sybil community detection algorithm can effectively detect the sybil community

around a sybil node with short running time.

- We propose two practical techniques to limit the number of attack edges in OSNs, and develop a Facebook application to demonstrate the feasibility of one of the techniques.

The rest of this chapter is organized as follows: in Section 4.1 we present the system model. In Section 4.2 we present the design of SybilDefender. The effectiveness of SybilDefender is shown experimentally in Section 4.3, and in Section 4.4 we conclude the chapter.

4.1 System Model

We denote the social network as a graph G consisting of vertices V and edges E. There are n honest users in the social network, each with one identity, denoted as an *honest node* in V. There are also one or more malicious users in the social network, each with a number of sybil identities. Each sybil identity is denoted as a *sybil node* in V. A relationship between two identities in the social network is represented as an edge connecting the two corresponding nodes in G. The edges in G are undirected. We name the edge between a sybil node and an honest node an *attack edge*. The *sybil region* consists of all the sybil nodes, while the *honest region* consists of all the honest nodes. All the sybil nodes are controlled by an adversary. Thus the adversary can create arbitrary edges within the sybil region.

SybilDefender is built upon the following assumptions:

The honest region is fast mixing. Fast mixing means a random walk of length $\Theta(\log n)$ is long enough such that with probability at least $1-\frac{1}{n}$, the last traversed

node is drawn from the node stationary distribution of the graph [112]. Generally speaking, random walks in a fast mixing graph converge quickly to the stationary distribution. The stationary distribution is a probability distribution $\overline{\pi}$ for V such that $\overline{\pi} = \overline{\pi}P$, where P is the transition matrix of the random walk process [74]. At each step of the random walk, the transition probability from node i to j is $P_{ij} = \frac{A_{ij}}{d_i}$, where d_i is the degree of node i. $A_{ij} = 1$ if i and j are connected, otherwise $A_{ij} = 0$. It can be easily proved that $\overline{\pi}_i$, the stationary probability of node i, is equal to $\frac{d_i}{2|E|}$. Yu et al. have shown that the real-world social networks are fast mixing [111]. The previous sybil defense schemes [33,111,112] are also built upon this assumption.

One known honest node. As previous schemes [33, 111, 112], we assume that there is at least one known honest node in the social network. This node is the starting point of our sybil identification algorithm.

The administrator knows the social network topology. This means that SybilDefender is a centralized sybil defense mechanism. Considering that all the current OSNs are under centralized control, it is natural for the administrators of these networks to take charge of mitigating sybil attacks.

The size of the sybil region is not comparable to the size of the honest region. Given the large user base of the current OSNs (Facebook (over 500 million), Twitter (over 200 million), Orkut (over 120 million)), it is reasonable to assume that the adversary cannot create such many sybil identities, especially considering that signing up a new user account always includes verifying an email address, providing some personal information, and solving CAPTCHAs.

The number of attack edges is limited. As a result, when the adversary creates many sybil nodes, there will be a disproportionately small cut between the honest region and the sybil region. The existence of a small cut disturbs

the fast-mixing property: the mixing between the honest nodes is fast, while the mixing between the honest nodes and the sybil nodes is slow. Previous schemes limit the number of attack edges by assuming that the honest users only establish links with their real-world friends [33,98,111,112], which has been shown to not hold in OSNs. The experiment by Bilge et al. [17] shows that on Facebook, the acceptance rate of friendship requests from a bogus account is around 20%. If an adversary launches a sybil attack, all the links created in this way are attack edges. We will address this problem in Section 4.2.4.

4.2 SybilDefender Design

SybilDefender consists of three components: a sybil identification algorithm, a sybil community detection algorithm, and two supporting approaches to limiting the number of attack edges. The three components can be used in conjunction to best mitigate sybil attacks. The task of the sybil identification algorithm presented in Section 4.2.1 is to determine whether a suspect node is sybil. Then we show how to efficiently detect the sybil community around a sybil node with our sybil community detection algorithm presented in Section 4.2.2. The reason why we need the second algorithm is that simply examining all the nodes in the social graph to find the sybil community is impractical. In Section 4.2.3, we present a Combo algorithm that combines the sybil identification algorithm with the sybil community detection algorithm. Finally, both algorithms are built upon the assumption that the number of attack edges is limited. In Section 4.2.4 we propose two approaches to supporting this assumption in online social networks.

4.2.1 Sybil Identification Algorithm

In this subsection we present a sybil identification algorithm that takes the social graph $G(V, E)$, a known honest node h, and a suspect node u as inputs, and outputs whether u is sybil or not. Our algorithm is based on random walks. A random walk on a graph is defined by the sequence of moves of a particle between nodes of G. If the particle is at node i with degree d_i, then the probability that the particle follows the edge (i, j) and moves to a neighbor j is $1/d_i$.

The intuition of our sybil identification algorithm is that, as there is a small cut between the honest region and the sybil region, the random walks originating from a sybil node tend to get ``trapped" into the sybil region. Also, since we assume that the size of the sybil region is not comparable to the size of the honest region, the number of nodes traversed by the random walks originating from an honest node will be larger than the number of nodes traversed by the random walks originating from a sybil node, as long as the random walks are long enough and we perform the random walks many times. For simplicity, we define the number of times one node being traversed by a set of random walks as the *frequency* of that node. Note that one node may be traversed by the same random walk multiple times.

The sybil identification algorithm consists of two phases, Algorithm 1 and Algorithm 2. The first phase takes G and h as inputs, and outputs the thresholds used by the second phase to identify sybil nodes. It only needs to be invoked once for each social network topology. As shown in Algorithm 1, the algorithm first performs f short random walks with length $l_s = \log n$ originating from the known honest node h. The f ending nodes are drawn from the node stationary distribution of the honest region, since we assume that the honest region is fast

Algorithm 1 PreProcessing(G, h)

```
1: J = {h}
2: for i = 1 to f do
3:    Perform a random walk with length l_s = log n originating from h
4:    J = J ∪ {the ending node of the random walk}
5: end for
6: l = l_min
7: while l <= l_max do
8:    for i = J.first() to J.last() do
9:       Perform R random walks with length l originating from node i
10:      Get n_i as the number of nodes with frequency no smaller than t
11:   end for
12:   output ⟨l, mean({n_i : i ∈ J}), stdDeviation({n_i : i ∈ J})⟩
13:   l = l + 100
14: end while
```

mixing. Following the proof in [113], the ending nodes are all honest nodes with high probability. After this the known honest node h and the f ending nodes are treated as *judge nodes*, from which the algorithm sets up the criterion to identify sybil nodes. Note the possibility that sybil nodes may exist in the group of the judge nodes does not influence the effectiveness of the algorithm, due to their very limited number. Starting from a minimum length l_{min} to a maximum length l_{max}, with an interval of 100 hops, for each length l, the algorithm performs R (ranging from 1000 to 2000 in our evaluation) random walks originating from every judge node, and counts the number of nodes whose frequency is no smaller than a threshold t, which is a small constant (5 in our evaluation). The algorithm collects $f + 1$ such values for each length l. Then it computes the mean and standard deviation of the $f + 1$ values and outputs a tuple as $\langle l, mean, stdDeviation \rangle$.

As shown in Algorithm 2, to determine whether a suspect node u is sybil, the algorithm first performs R random walks with an initial length $l = l_0$ originating from u. l_0 is larger than or equal to l_{min} used in Algorithm 1. The algorithm

Algorithm 2 SybilIdentification($G, u, tuples\ from\ Alg.1$)

1: $l = l_0$
2: **while** $l <= l_{max}$ **do**
3: Perform R random walks with length l originating from u
4: m = the number of nodes whose frequency is no smaller than t
5: Let the tuple corresponding to length l in the outputs of Algorithm 1 be $\langle l, mean, stdDeviation \rangle$
6: **if** $mean - m > stdDeviation * \alpha$ **then**
7: output u is sybil
8: end the algorithm
9: **end if**
10: $l = l * 2$
11: **end while**
12: output u is honest

then compares the number of nodes whose frequency is no smaller than t with the $mean$ value in tuple $\langle l, mean, stdDeviation \rangle$ outputted by Algorithm 1. If the former is smaller than the latter by an amount larger than $stdDeviation * \alpha$ ($\alpha = 20$ in our evaluation), we consider u is sybil and end the algorithm. Otherwise, the algorithm doubles l and repeats the process, until l is larger than l_{max}. If u is still not identified as sybil when the value of l reaches l_{max}, we consider it honest and end the algorithm.

Given a social graph $G(V, E)$ and a known honest node h, l_{max}, the maximum random walk length that decides when to end the algorithm, can be determined as follows. We do R random walks originating from h with length l_{max}. The number of nodes with frequency no smaller than t should be larger than $|V|/2$. Given that we assume the sybil region is smaller than the honest region, l_{max} determined in this way is large enough for R random walks originating from a sybil node to cover the sybil region, so as to exhibit the difference between the random walks originating from an honest node and from a sybil node. Our algorithm adaptively tests the suspect node while doubling the random walk length

Table 4.1: Notations used in the analysis

$G(V, E)$	social graph, V is the set of nodes, E is the set of edges
P	transition matrix of the random walk process
n	number of honest nodes in G
λ	initial state vector of the random walk process
$\overline{\pi}$	stationary distribution of G
l	random walk length
Q_l	accumulated probability distribution of the nodes being traversed by a random walk with length l
t	threshold frequency used in the sybil identification algorithm
R	number of random walks originating from a given node
$\mathcal{D}(d)$	number of nodes with degree d

each time. This guarantees that the algorithm can identify the sybil nodes in differently sized sybil regions: for small sybil regions short random walks are already enough, while for large regions long random walks need to be performed, since the footprint of short random walks in a large sybil region may be similar to that in the honest region.

4.2.1.1 Analysis of the Sybil Identification Algorithm

In this subsection we investigate the validity of our sybil identification algorithm with theoretical analysis. For the ease of analysis we list the used notations in Table 4.1. A random walk with length l on an undirected graph G can be modeled as a Markov Chain process. The starting state of the random walk is described as λ, the initial state vector of V. $\lambda_v = 1$ if v is the starting node of the random walk, otherwise $\lambda_v = 0$. As defined in Section 4.1, P is the transition matrix of the random walk process. Therefore, the probability distribution of the nodes being visited by the i^{th} hop of the random walk is λP^i. Based on our fast-mixing assumption, λP^i converges to the stationary distribution $\overline{\pi}$ of G with

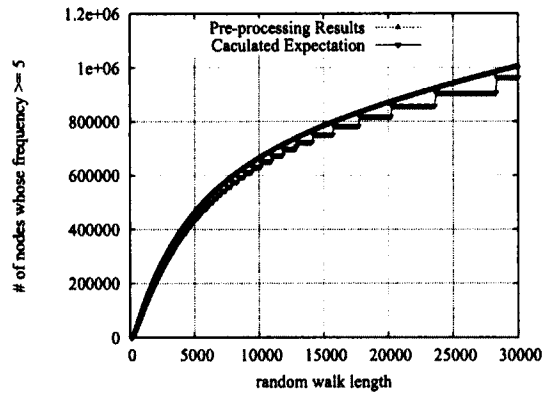


Figure 4.1: Pre-processing results and calculated expectation values

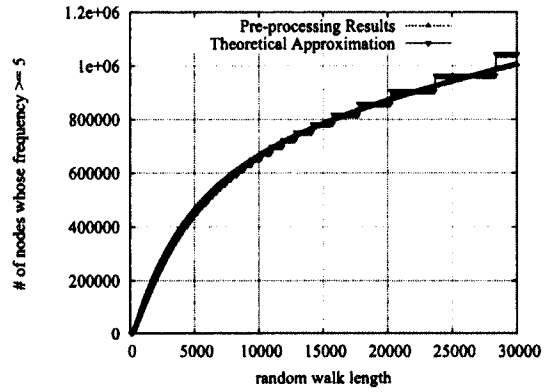


Figure 4.2: Pre-processing results and theoretical approximate results

$i \geq \Theta(\log n)$. The accumulated probability distribution of nodes being traversed by a random walk with length l is $Q_l = \sum_{i=0}^{l} \lambda P^i$, and $(Q_l)_j$, the j^{th} element in vector Q_l, is the expected number of times node j being traversed by a random walk with length l. Therefore, $R \cdot (Q_l)_j$ is the expected number of times node j being traversed by R random walks with length l originating from the same honest node, i.e., the expected frequency of j.

The pre-processing phase of our sybil identification algorithm sets up the criterion to identify sybil nodes by performing R random walks originating from each judge node with every length value $l \in \{l_{min}, l_{min} + 100, ..., l_{max}\}$, respectively, and records the mean and the standard deviation of the number of tra-

versed nodes with frequency no smaller than t. Define set S_l as $\{j|R\cdot(Q_l)_j \geq t\}$, then $|S_l| = |\{j|\sum_{i=0}^{l} R\cdot(\lambda P^i)_j \geq t\}|$ is the expected number of nodes whose frequency is no smaller than t with R l-hop random walks. With a randomly chosen source node and $R = 2000$, based on our Facebook data set, we calculate $|S_l|$ for different lengths and draw the *calculated expectation* curve in Figure 4.1. To demonstrate the validity of our sybil identification algorithm, we set the number of judge nodes to be 10 and draw the *pre-processing results* curve based on the mean value outputs of the pre-processing phase in the same figure. It shows that even with a small number of judge nodes, the two curves match well when the random walk length is smaller than 10000 hops. As the random walk length increases there shows some horizontal segments in the calculated expectation curve. This is because in a fast-mixing network, $(\lambda P^i)_j$ converges to $\frac{d_j}{2|E|}$ with $i \geq \Theta(\log n)$, where d_j is the degree of node j and E is the set of edges. This means that with $l \geq \Theta(\log n)$ the value of $\sum_{i=0}^{l}(\lambda P^i)_j$ for all the nodes with the same degree increases by the same amount when l increases by 1, and thus their expected frequency, $\sum_{i=0}^{l} R\cdot(\lambda P^i)_j$, will reach the threshold t at the same random walk length, which leads to the jumps in the calculated expectation curve. Note that although the calculated expectation curve is divided into horizontal segments when the random walk length is large, its inflection points still match well with the pre-processing results curve. Figure 4.1 illustrates that *with a small number of judge nodes and limited* R, the results derived from the pre-processing phase of the sybil identification algorithm are already accurate enough to match with the expectation values.

Moreover, we will show that the results derived from the pre-processing phase *starting from a random honest node* are generic enough to serve as the criterion to identify sybil nodes. Since the network is fast mixing, given a random

starting node, i.e., a random initial state vector λ, we have

$$\begin{aligned} Q_l &= \sum_{i=0}^{l} \lambda P^i \\ &\approx \lambda + \lambda P + ... + \lambda P^{\Theta(\log n)-1} + \underbrace{\overline{\pi} + ... + \overline{\pi}}_{l-\Theta(\log n)+1} \\ &\approx l\overline{\pi}. \end{aligned}$$

Besides, $\overline{\pi}_j = \frac{d_j}{2|E|}$, then we have $(Q_l)_j \approx l\frac{d_j}{2|E|}$. Recall that $R \cdot (Q_l)_j$ is the expected frequency of node j. To make this value no smaller than t, we have

$$R \cdot (Q_l)_j \approx R \cdot l\frac{d_j}{2|E|} \geq t \Rightarrow d_j \geq \frac{2t|E|}{lR}. \tag{4.1}$$

Define $S_l' = \{j | d_j \geq \frac{2t|E|}{lR}\}$. Then $|S_l'|$ is the approximate number of nodes whose frequency is no smaller than t with R l-hop random walks. Let $\mathcal{D}(d)$ be the number of nodes with degree d. Then

$$|S_l'| = \sum_{d=\lceil \frac{2t|E|}{lR} \rceil}^{\max} \mathcal{D}(d). \tag{4.2}$$

Following Equation 4.2 we draw the *theoretical approximation* curve in Figure 4.2 based on our Facebook data set, and we compare it with the pre-processing results curve identical to that in Figure 4.1. Note that Equation 4.2 is irrelevant to the initial state vector λ, so the shape of the theoretical approximation curve does not rely on the starting node. Figure 4.2 shows that the pre-processing results also match well with the theoretical approximate results. Similar to the caculated expectation curve, there are horizonal segments in the theoretical approximation curve. This is because node degrees are integers and l needs to increase by a certain amount such that the value of $\frac{2t|E|}{lR}$ reaches the next integer.

Nevertheless, the middle point of each horizonal segment still matches with the pre-processing results curve. Figure 4.2 illustrates that the pre-processing results drawn from a random honest node can be effectively used as the criterion to identify sybil nodes.

To gain an understanding of the difference between the footprint of random walks originating from an honest node and from a sybil node, assume φ is the expected number of hops for the random walks starting from a sybil node to enter the honest region. If we draw the curve for the number of nodes with frequency no smaller than t based on the random walks starting from that sybil node, it is approximately like moving the pre-processing results curve in Figures 4.1 and 4.2 to the right by φ and then raising it by the size of the sybil region. In the evaluation we will show that this difference is large enough to identify the sybil nodes.

4.2.2 Sybil Community Detection Algorithm

After one sybil node is identified, our sybil community detection algorithm can be used to detect the sybil community surrounding it. The sybil community detection algorithm takes the social graph $G(V, E)$ and a known sybil node s as inputs, and outputs the sybil community around s. The sybil node s can be identified by our sybil identification algorithm or any previous scheme. We define a sybil community as a subgraph of G consisting of only sybil nodes, and there is no small cut in this subgraph. The reason why we make this definition is that if a small cut does divide the sybil region into two parts S_1 and S_2, and the known sybil node s is in S_1, then, from the point of view of s, the honest region and S_2 are similar, since there is already a small cut between S_1 and the honest

Algorithm 3 WalkLengthEstimation(G, s)

```
1: l = l_0/2
2: deadWalkRatio = 0
3: while deadWalkRatio < β do
4:    l = l * 2
5:    deadWalkNum = 0
6:    for i = 1 to R do
7:       Perform a partial random walk originating from s with length l
8:       if the partial random walk is dead before it reaches l hops then
9:          deadWalkNum++
10:      end if
11:   end for
12:   deadWalkRatio = deadWalkNum / R
13: end while
14: output l
```

region and also a small cut between S_1 and S_2. When there is a small cut in the sybil region, our algorithm can detect the sybil community s belongs to.

Our algorithm relies on performing *partial* random walks originating from s. Each partial random walk behaves the same as the *standard* random walks used in the previous subsection, except that it does not traverse the same node more than once. Therefore, when a partial random walk reaches a node with all the neighbors traversed by itself, this partial random walk is ``dead" and cannot proceed. This property makes a partial random walk originating from a sybil node less likely to leave the sybil region, compared with a standard random walk, since many such walks ``die" when they hit the border of the sybil region. Similar to the sybil identification algorithm, the intuition behind this algorithm is that the partial random walks originating from a sybil node tend to be trapped within the sybil region, and thus we can detect the sybil community by examining the nodes traversed by the partial random walks.

The sybil community detection algorithm consists of two phases, Algorithm

3 and Algorithm 4. The task of Algorithm 3 is to estimate the needed length of the partial random walks used in Algorithm 4. Starting from an initial length l_0, the algorithm performs R partial random walks originating from s and counts the ratio of dead walks, which are the walks that cannot proceed before they reach the required length. If this ratio is smaller than β, a threshold close to 1 (0.95 in our evaluation), the algorithm doubles the current length and performs the partial random walks again. This process is repeated until the dead walk ratio is no smaller than β. Then the algorithm outputs the current random walk length l. The reasoning is that the number of untraversed sybil nodes is very small (often equals to 0 in our evaluation) when the dead walk ratio is close to 1 and with a relatively large R (2000 in our evaluation).

Algorithm 4 takes G, s, and the estimated length l as inputs and outputs the sybil community surrounding s. The reason why we need Algorithm 4 is that not all the nodes traversed by the partial random random walks in Algorithm 3 are sybil nodes, as some walks pass the small cut and enter the honest region, and we need an algorithm to select the sybil nodes from the set of traversed nodes. To achieve this, Algorithm 4 leverages a metric called *conductance* [57], defined as follows. Let d be the sum of the degrees of all the nodes in set S, and a be the number of edges with one endpoint in S and one endpoint in $\overline{S}$. Then the conductance of S is a/d. The conductance of a set S measures the quality of the cut between S and $\overline{S}$: the smaller the conductance is, the smaller the cut is. Since we assume that there is a small cut between the honest region and the sybil region, using conductance as the objective of the greedy algorithm fits the problem well. In this algorithm we let the conductance of an empty set be 1.

Algorithm 4 runs by first performing R partial random walks originating from the known sybil node s, with the length decided by Algorithm 3. Then the al-

Algorithm 4 SybilRegionDetection(G, s, l *from Alg.*3)

```
1: Set the frequency of all the nodes to be 0
2: for i = 1 to R do
3:    Perform a partial random walk originating from node s with length l
4:    s.frequency++
5:    for j = 1 to l do
6:       Let the j^th hop of the partial random walk be node k
7:          k.frequency++
8:    end for
9: end for
10: traversedList = Sort the traversed nodes by their frequency in decreasing
    order
11: counter = 0
12: S = ∅
13: do
14:    counter = conductance(S)
15:    for i = traversedList.first() to traversedList.last() do
16:       if node i ∈ S then
17:          continue
18:       end if
19:       if conductance({i} ∪ S) <= conductance(S) then
20:          S = {i} ∪ S
21:       end if
22:    end for
23: while (counter > conductance(S))
24: output S
```

gorithm sorts all the traversed nodes by their frequency in decreasing order. Starting from the first node, which is always s, the algorithm iterates the sorted list and adds the encountered node to set S if doing so does not increase the conductance of S. After all the nodes in the sorted list are examined, the algorithm records the current conductance value, starts a new iteration from the top of the list and examines each node that is not in S. This process is repeated until the conductance value stays the same at the end of two consecutive iterations. Then the algorithm outputs S as the detected sybil community. The intuition is that by performing the partial random walks originating from a sybil node with

suitable length many times, the sybil community surrounding the sybil node is covered by the partial random walks. Also, the sybil nodes tend to be in front of the honest nodes in the sorted list, since a large number of partial random walks cannot enter the honest region, due to the existence of the small cut between the honest region and the sybil region. As a result, the greedy algorithm will first try to add the nodes that are more likely sybil to S. This algorithm only relies on performing R partial random walks originating from a sybil node, which makes it very efficient and scalable to large-sized social networks.

4.2.3 Combine Sybil Identification with Sybil Community Detection

Our sybil identification algorithm takes as input a suspect node, and outputs if the suspect node is sybil. In comparison, our sybil community detection algorithm takes as input a sybil node, and outputs the sybil community surrounding the sybil node. Each algorithm consists of a preparation phase and a detection phase. In this subsection we consider how to efficiently combine the two algorithms, such that by running the Combo algorithm once, the administrator is able to learn if the suspect node is sybil, and if it is, the sybil community around it.

The Combo algorithm consists of two phases, Algorithm 1 and Algorithm 5. Algorithm 1 only needs to be run once for each target social graph. Algorithm 5 takes as input a suspect node u, the tuples from Algorithm 1, and the social graph G, and it outputs u's identity (sybil or honest) as well as the sybil community surrounding u (if sybil). The intuition of the Combo algorithm is that, if we find a sybil node, instead of performing partial random walks as in Section

Algorithm 5 Combo($G, u, tuples\ from\ Alg.1$)

```
1: l = l_0
2: while l <= l_max do
3:    Perform R random walks with length l originating from u
4:    m = the number of nodes whose frequency is no smaller than t
5:    Let the tuple corresponding to length l in the outputs of Algorithm 1 be
      ⟨l, mean, stdDeviation⟩
6:    if mean − m > stdDeviation * α then
7:       output u is sybil
8:       traversedList = Sort the traversed nodes by their frequency in decreas-
         ing order
9:       counter = 0
10:      S = ∅
11:      do
12:         counter = conductance(S)
13:         for i = traversedList.first() to traversedList.last() do
14:            if node i ∈ S then
15:               continue
16:            if conductance({i} ∪ S) <= conductance(S) then
17:               S = {i} ∪ S
18:      while (counter > conductance(S))
19:      output S
20:      end the algorithm
21:   end if
22:   l = l * 2
23: end while
24: output u is honest
```

4.2.2 to detect the surrounding sybil community, we directly analyze the simple random walks derived in the identification method. The analysis is based on the conductance measure. We sort the nodes by their frequency, the number of times being traversed by the simple random walks, in decreasing order, and iteratively add nodes to the detected sybil set, until the conductance value stays the same at the end of two consecutive iterations.

It is easy to see that the Combo algorithm behaves the same as the sybil identification algorithm (Section 4.2.1) when identifying sybil nodes, while it diverges from the sybil community detection algorithm (Section 4.2.2) in that it

reuses the simple random walks performed in the identification phase to search for the sybil community. The advantage is that it avoids estimating the partial random walk length (Algorithm 3) and performing partial random walks (Algorithm 4), and thus incurs much smaller computation overhead. However, compared with partial random walks, the simple random walks originating from a sybil node are more likely to escape the sybil region. Our evaluation in Section 4.3 shows that replacing partial random walks with simple random walks slightly impact detection accuracy, while it significantly reduces running time of the algorithm. Therefore, the Combo algorithm provides a tradeoff between efficiency and accuracy: to detect sybil nodes, users can use the Combo algorithm if running time is a concern. Otherwise, they can use the stand-alone sybil identification and sybil community detection algorithms if detection accuracy is more important.

4.2.4 Limiting the Number of Attack Edges

Our algorithms rely on the assumption that the number of attack edges is limited. However, it has been shown that not all the relationships in OSNs are trusted [17]. In this subsection we propose two approaches to limiting the number of attack edges in OSNs.

4.2.4.1 Relationship Rating

One approach to limiting the number of attack edges in these networks is to allow the users to rate their relationships. To demonstrate this we develop a Facebook application named *Rate Your Relationships*, as shown in Figure 4.3. The users of the application can rate each of their relations on Facebook either

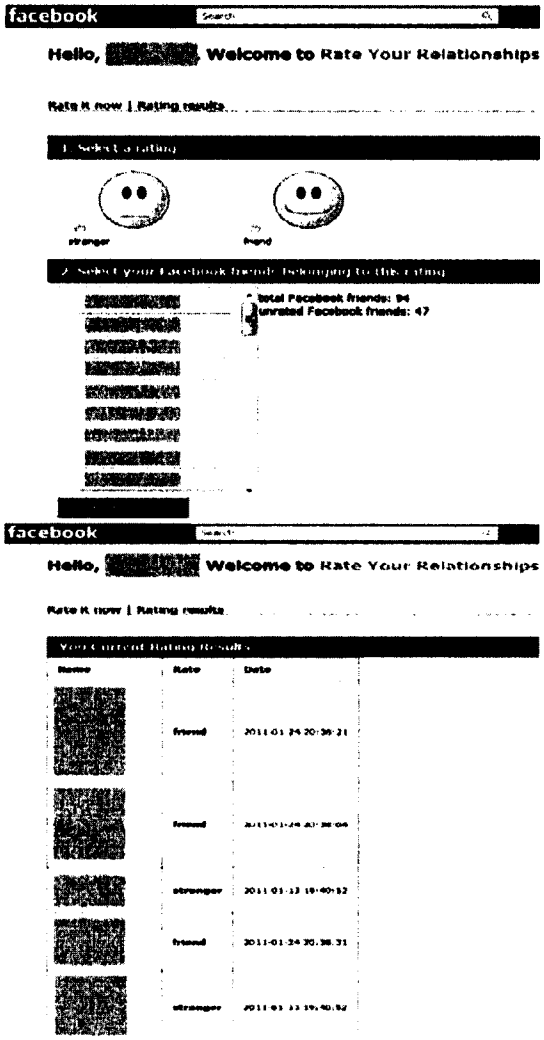


Figure 4.3: Our Facebook application: Rate Your Relationships

as ``Friend" or ``Stranger", where ``Stranger" means the user hardly has any impression about this relation. In this way, the user's relations are classified into two categories. The number of attack edges can be limited by removing the relationships rated as stranger from the social graph when applying the sybil defense schemes. The rationale is that even if an adversary can create many links between the sybil identities and the honest identities, it is hard for him to convince the honest users that those sybil identities are their acquaintances.

4.2.4.2 Activity Network

We can also use the concept of *activity network* [30,100,104] to limit the number of attack edges. Activity network is a network graph that is based on the interaction between users, rather than mere relationship. It contains all the nodes from its social network counterpart, but only a subset of edges. Two nodes share an edge in an activity network if and only if they have interacted directly through the communication mechanisms or applications provided by the corresponding social network. In other words, a social network is transformed into an activity network by removing the weak connections with no user activity. If the sybil defense schemes leverage the topologies of the activity networks, the number of attack edges an adversary can create can be further limited.

4.3 Evaluation

4.3.1 Data Sets and Experiment Setup

In this section we evaluate the effectiveness of SybilDefender using two data sets [73, 104] from Orkut and Facebook, respectively. The Orkut data set consists of 3,072,441 nodes and 117,185,083 edges, with an average degree of 76.28, while the Facebook data set consists of 3,097,165 nodes and 28,377,481 edges, with an average degree of 18.32. To the best of our knowledge, these are the largest data sets that have ever been used in evaluating the sybil defense schemes that leverage social network topologies. The reason why the average degree of the Facebook data set is much smaller than that of the Orkut data set is that the Orkut data set is a breadth-first sample of the Orkut social graph, which maintains the topological properties of Orkut like average degree;

on the other hand, the Facebook data set is a regional network in Facebook. Two nodes share an edge in this data set if and only if both of them are members of the same regional network, and they are Facebook friends with each other. By evaluating the performance of SybilDefender on these two data sets, we show that SybilDefender applies to social networks with different topological properties.

In the experiments we use two models to construct the sybil regions respectively: the preferential attachment (PA) model [11] and the Erdös-Rényi (ER) model [37]. Both models are widely used in network analysis. The networks constructed with the PA model are scale free, which means their node degrees follow a power-law distribution, a well accepted property of social networks [10, 73, 88, 104]. The PA model has been used in previous research to build the sybil regions [33, 101]. The topologies built through the ER model, on the other hand, are random networks with no particular bias, which emulate the arbitrary structures of the sybil regions. In our experiments, to build a sybil region and connect it to a real-world social network sample, we follow the suggestion by Yu et al. [112] that the most effective way for an adversary to launch a sybil attack is to first compromise a small number of existing nodes, so as to quickly increase the number of attack edges. We first randomly select nodes from the data set to be compromised nodes, until the number of edges between the compromised nodes and the other nodes is g_0, which is the number of attack edges. The compromised nodes are all sybil nodes. They introduce γ additional sybil nodes, and establish a connected scale-free topology through the PA model, or a connected random topology through the ER model among all the sybil nodes. We label all the other nodes in the data set as honest nodes. The average degree of the sybil region built with the PA model is set to be equal to

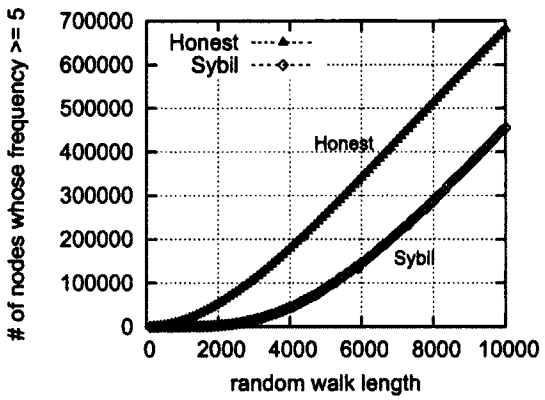


Figure 4.4: Difference between the coverage of random walks originating from honest nodes and from sybil nodes

the average degree of the corresponding data set, while the average degree of the sybil region built with the ER model ranges from 8 to 11, representing a sparse topology compared with the realistic social networks. Note that in the evaluation of some previous schemes, the social network samples are first pre-processed by removing the nodes with small degrees [33, 111], to prevent such nodes from degrading the effectiveness of these schemes. Instead, we do not make any modification on the published data sets.

4.3.2 Evaluation of the Sybil Identification Algorithm

Yu et al. [111] proved that for all the sybil defense mechanisms that leverage the fast-mixing property, the number of admitted sybil nodes per attack edge is lower bounded by $\Omega(1)$. The rationale is that the fasting mixing property of the network is not disrupted if each attack edge introduces few sybil nodes. In this subsection we will show that the performance of our sybil identification algorithm approaches this theoretical bound, and our algorithm outperforms SybilLimit by one to two orders of magnitude in both accuracy and running time.

Table 4.2: 10 sybil nodes per attack edge (10000 sybil nodes) false positive and negative rates

	Orkut				Facebook			
	PA Model		ER Model		PA Model		ER Model	
	F^+	F^-	F^+	F^-	F^+	F^-	F^+	F^-
1000RWs	0	0	0	0.11%	0	0.07%	0.1%	0.16%
1500RWs	0	0.01%	0	0.11%	0.4%	0.08%	0.2%	0.1%
2000RWs	0	0	0	0.04%	0.3%	0.1%	0.5%	0.1%

The intuition of our sybil identification algorithm is that, because of the existence of a small cut between the honest region and the sybil region, there is a difference between the coverage of random walks originating from an honest node and from a sybil node. Figure 4.4 illustrates this difference. Here, we use the PA model to construct the sybil region. We set the size of the sybil region to be 10000 nodes, and the number of attack edges to be 1000. In the experiments we perform 1000 random walks originating from each randomly selected source node. The upper curve in Figure 4.4 is the number of nodes traversed by random walks originating from an honest node no smaller than 5 times, while the lower curve is the number of nodes traversed by random walks originating from a sybil node no smaller than 5 times. Each point in the curves represents the mean value of 20 experiments. It is easy to see that the difference is larger than 200,000 nodes when the random walk length reaches 10000 hops. As described in Algorithm 2, we use $\tau = mean - \alpha * stdDeviation$ as the threshold to identify sybil nodes. In our experiments we observe that $stdDeviation < 1500$, so the sybil nodes can be identified even with a relatively large α, to limit the number of falsely identified honest nodes.

To evaluate our sybil identification algorithm, the parameters we used in the experiments are as follows: $l_{min} = 100$, $l_{max} = 10000$, $l_0 = 1000$, $t = 5$, $\alpha = 20$, $l_s = 20$, $f = 100$, $R \in \{1000, 1500, 2000\}$. When building the sybil regions, we set

Table 4.3: 5 sybil nodes per attack edge (5000 sybil nodes) false positive and negative rates

	Orkut				Facebook			
	PA Model		ER Model		PA Model		ER Model	
	F^+	F^-	F^+	F^-	F^+	F^-	F^+	F^-
1000RWs	0	0.02%	0	0.28%	0	0.22%	0.1%	0.54%
1500RWs	0	0.02%	0	0.32%	0.3%	0.12%	0.2%	0.44%
2000RWs	0	0	0	0.22%	0.5%	0.04%	0.5%	0.4%

the number of attack edges to be 1000. We define the *false positive rate* as the percentage of the honest nodes identified to be sybil, and the *false negative rate* as the percentage of the sybil nodes identified to be honest. In the experiments we obtain the false positive and negative rates of our algorithm. As we use large-scale topologies in the experiments, it is infeasible to examine all the honest nodes to get the exact false positive rate. To estimate the false positive rate of the algorithm, in each experiment we randomly select 1000 honest nodes as suspects and use our sybil identification algorithm to test them. To get the false negative rate, in each experiment we use our algorithm to test every sybil node. In the experiments we vary the number of sybil nodes per attack edge. For each value we evaluate the algorithm on two real-world topologies, using two sybil region construction models, and with three values of R, the number of random walks performed in the algorithm, respectively.

Table 4.2 shows the results when each attack edge introduces 10 sybil nodes. It is easy to see that our algorithm achieves very low false positive and negative rates in all the cases. We find that all the sybil nodes that cannot be correctly identified are compromised nodes, as they are on the small cut between the honest region and the sybil region. Similarly, all the falsely identified honest nodes are close to the small cut.

Table 4.3 shows the results when each attack edge introduces 5 sybil nodes.

Table 4.4: 1 sybil node per attack edge false positive and negative rates

	Orkut		Facebook	
	PA Model		PA Model	
	F^+	F^-	F^+	F^-
1000RWs	0	0.6%	0%	6.2%
1500RWs	0	0.7%	0.4%	4.4%
2000RWs	0	0.2%	0%	1.4%

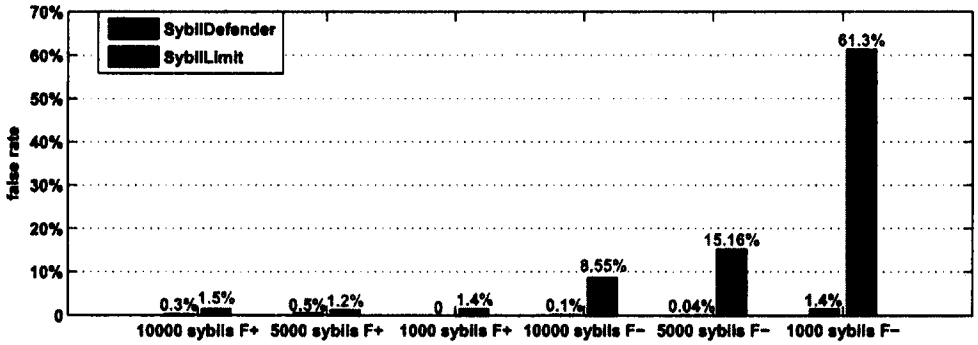


Figure 4.5: Comparison between the false positive and negative rates of SybilDefender and those of SybilLimit

The results are similar to those in Table 4.2. We observe that with the increase of the number of random walks performed in the algorithm, the false positive rate raises, while the false negative rate decreases. The reason is that the more random walks are performed, the smaller the standard deviation of the number of nodes whose frequency, the number of times being traversed, is no smaller than t is. As a result, the threshold τ increases when more random walks are performed, which makes it less likely for a sybil node to be identified as honest, and vice verse for honest nodes.

Table 4.4 shows the results when each attack edge introduces only one sybil node. The false negative rate for the Facebook data set is higher than the results shown in Table 4.3. This is because the difference between the coverage of the random walks originating from an honest node and from a sybil node becomes smaller compared with the cases when each attack edge introduces more sybil

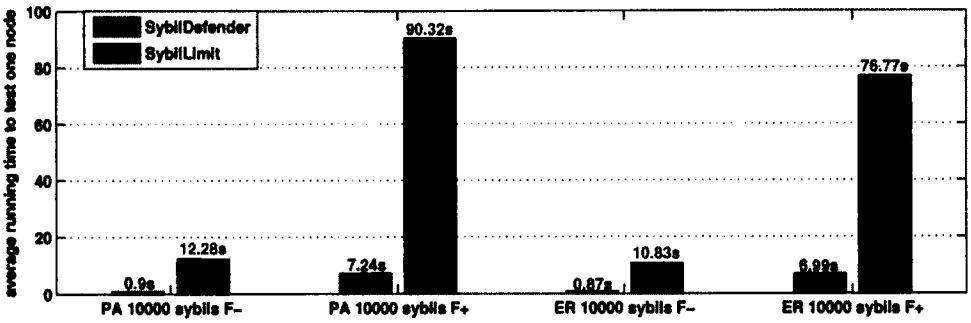


Figure 4.6: Comparison between the average running time to test one node by SybilDefender and that by SybilLimit

Table 4.5: 10 sybil nodes per attack edge (100000 sybil nodes) false positive and negative rates

	Orkut				Facebook			
	PA Model		ER Model		PA Model		ER Model	
	F^+	F^-	F^+	F^-	F^+	F^-	F^+	F^-
1000RWs	0	0.06%	0	0.13%	0.4%	0.76%	0.4%	0.78%
1500RWs	0	0.03%	0	0.12%	0.6%	0.67%	0.4%	0.68%
2000RWs	0	0.03%	0	0.12%	0.7%	0.62%	0.7%	0.66%

Table 4.6: 5 sybil nodes per attack edge (50000 sybil nodes) false positive and negative rates

	Orkut				Facebook			
	PA Model		ER Model		PA Model		ER Model	
	F^+	F^-	F^+	F^-	F^+	F^-	F^+	F^-
1000RWs	0	0.14%	0	0.23%	0.3%	1.40%	0.2%	1.31%
1500RWs	0	0.11%	0	0.22%	0.4%	1.31%	0.5%	1.07%
2000RWs	0	0.04%	0	0.21%	0.5%	1.09%	0.5%	0.97%

nodes. The experimental results show that our sybil identification algorithm can identify nearly all the sybil nodes when each attack edge introduces 10 or 5 sybil nodes, and an overwhelming majority of sybil nodes when each attack edge introduces 1 sybil node, both with very low false positive rate.

To investigate the performance of our sybil identification algorithm when more sybil nodes are controlled by the adversary, we raise the number of attack edges to 10000 and repeat the experiments. Following the approach mentioned

Table 4.7: False rates of SybilLimit on the Facebook data set

	PA Model		ER Model	
	F^+	F^-	F^+	F^-
10000 sybils	1.5%	8.55%	0.6%	15.35%
5000 sybils	1.2%	15.16%	1.4%	32.62%
1000 sybils	1.4%	61.3%	0.8%	85.3%

above, creating 10000 attack edges means that on average the adversary needs to compromise 131 honest nodes in the Orkut data set, or 546 honest nodes in the Facebook data set. Table 4.5 lists the experimental results when each attack edge introduces 10 sybil nodes, which leads to a sybil region consisting of 100000 sybil nodes; Table 4.6 lists the results when each attack edge introduces 5 sybil nodes. It is easy to see that our algorithm still achieves low false positive and negative rates in these scenarios.

4.3.2.1 Comparison with existing schemes

We fully implemented SybilLimit and evaluated it using our Facebook data set. We didn't evaluate SybilLimit on the Orkut data set as the running time is approximately 4 times longer, since the average degree of the Orkut data set is about 4 times of the average degree of the Facebook data set. Following the method in [111], we found the optimal parameters for SybilLimit on the Facebook data set. We set w, the length of random routes, to be 20 hops, and r, the number of instances of the random route generation protocol, to be 10000. Table 4.7 lists SybilLimit's false positive and negative rates when each attack edge introduces 10 sybil nodes, 5 sybil nodes, and 1 sybil node, respectively. The results show that with each attack edge introducing 10 sybil nodes, SybilLimit accepts 8.55% of the sybil nodes when the sybil region is constructed by the PA model, and 15.35% of the sybil nodes when the sybil region is constructed

by the ER model. In comparison, SybilDefender only accepts 0.1% of the sybil nodes in both cases when $R = 2000$. With the decrease of the number of sybil nodes introduced by each attack edge, the false positive and negative rates of SybilLimit raise significantly. When each attack edge introduces one sybil node, SybilLimit accepts the majority of the sybil nodes.

Figure 4.5 compares the false positive and negative rates of SybilDefender with those of SybilLimit, when the sybil region is built with the PA model. It is easy to see that in all the three cases the false positive rate of SybilDefender is lower than that of SybilLimit, and the false negative rate of SybilDefender is lower than that of SybilLimit by one to two orders of magnitude. The reason is SybilLimit assumes that almost all the short random routes originating from an honest node will stay within the honest region, and it bounds the number of admitted sybil nodes by the number of attack edges and random route length. When each attack edge introduces few sybil nodes, SybilLimit cannot effectively identify the sybil nodes. On the other hand, SybilDefender interprets the small cut between the honest region and the sybil region as a bias in the coverage of the random walks originating from an honest node and from a sybil node. It can effectively identify the sybil nodes even when the number of sybil nodes introduced by each attack edge approaches the theoretical lower bound.

Figure 4.6 compares the average running time to test one node on one core of an Intel Xeon 2.93GHz processor by SybilDefender with that by SybilLimit. The results show that SybilDefender is faster than SybilLimit by more than 10 times. The reason is that SybilLimit invokes a large number ($r = 10000$ for our Facebook data set) of instances of the random route generation protocol [111]. Within each instance a random routing table is generated for every node in the social graph. After all the instances are finished, SybilLimit verifies if the inter-

section condition and the balance condition are satisfied to determine whether to accept each suspect node. By contrast, SybilDefender only relies on performing a limited number of random walks, which can be done in a short time even on large-scale network graphs.

Viswanath et al. proposed using a community detection algorithm [101] as the ranking algorithm to investigate the similarity between different sybil defense schemes. We evaluated their algorithm using our two data sets, and found that the algorithm alone cannot be used to identify the sybil nodes. The reason is that the algorithm starts from an honest node and iteratively adds nodes that improves the normalized conductance at each step. In our evaluation the normalized conductance always reaches the first inflection point after adding only several honest nodes. As a result, their algorithm cannot distinguish the sybil nodes from the honest nodes without providing a cutoff point.

We also evaluated Gatekeeper [98] using our data sets, which heavily relies on the assumption that the social networks are random expander. This assumption is stronger than our fast-mixing assumption and has not been validated in previous research, which makes Gatekeeper suffer from high false positive and negative rates on the real-world social topologies that exhibit asymmetries. For example, on our Facebook data set with a 10000-node sybil region built through the PA model, the average false positive rate of Gatekeeper is 11.7%, and the average false negative rate is 17.2%. When the sybil region is built with the ER model, the average false positive rate is 11.7%, and the average false negative rate is 14.7%. In the evaluation we used the parameters ($m = 100, f_{admit} = 0.2$) recommended by [98] and repeated each experiment 20 times.

Table 4.8: False rates of the sybil identification algorithm on a weighted social network

10000 sybils				5000 sybils			
PA Model		ER Model		PA Model		ER Model	
F^+	F^-	F^+	F^-	F^+	F^-	F^+	F^-
0.7%	0.15%	0.5%	0.21%	0.7%	0.20%	0.7%	0.56%

4.3.2.2 Evaluation of the Sybil Identification Algorithm on a Weighted Social Network

As mentioned in Section 4.2.4, activity network is a social network model that takes the activities between users into account. Similarly, given the user activity information, a social network can be transformed into a weighted graph, by assigning a weight to each edge based on the number of interactions between the two endnodes. There are also other forms of weighted social networks, e.g., the trust networks whose edge weights denote the level of trust [8], and the networks of co-authorship where an edge weight is the number of papers co-authored by the two endusers [78].

To investigate the performance of our sybil identification algorithm on such weighted social networks, we use the data set provided by Wilson et al. [104]. The data set is an undirected, weighted graph. It has the same set of nodes and the same topology as the Facebook data set used in previous evaluations. However, each edge in the graph is assigned a weight, based on the number of interactions (wall posts and photo comments) between the two endnodes. For example, assuming user i and user j are friends in the date set, if the number of interactions between i and j is 10, then the weight of edge $\overline{ij}$ is 10. Otherwise, if there is no interaction between i and j ever, then the weight is 0. In total, this weighted graph records 17,644,327 interactions between 3,097,165 users,

which means that on average each user takes part in 11.4 interactions.

In our sybil identification algorithm, at each step of a random walk, the transition probability from node i to j is $P_{ij} = \frac{A_{ij}}{d_i}$, where d_i is the degree of node i. $A_{ij} = 1$ if i and j are connected, otherwise $A_{ij} = 0$. In other words, at each intermediate node of a random walk, when selecting the next hop, all its neighbors are treated equally. To apply this algorithm on the weighted social networks, we take the edge weights into account when choosing the next hop, and define *weighted* random walks. At each step of a weighted random walk, the transition probability from node i to j is $P_{ij} = \frac{A_{ij}(1+w_{ij})}{d_i+w_i}$, where w_{ij} is the weight of edge $\overline{ij}$, d_i is the degree of node i, and w_i is the sum of weights of all the edges connecting node i. $A_{ij} = 1$ if i and j are connected, otherwise $A_{ij} = 0$. As a result, the higher weight an edge has, with higher possibility the weighted random walk will traverse that edge. The modified sybil identification algorithm runs by performing weighted random walks.

Table 4.8 shows the false rates of the modified sybil identification algorithm on the weighted social network sample. The number of attack edges is 1000, $R = 2000$, and the parameters in the algorithm are the same with the ones used in previous evaluations. When constructing the sybil regions, we randomly assign weights to the edges connecting sybil nodes, such that the average weight is equal to the average weight of our data set. The results illustrate that our algorithm achieves similar performance on the weighted graph: both false positive and negative rates remain low. It is our ongoing work to investigate whether we can derive new algorithms that can further improve the performance.

Table 4.9: False rates of the sybil identification algorithm operating with trust-driven random walks

	lazy random walk			similarity-based random walk		
# of sybils	10000	5000	1000	10000	5000	1000
F^+	0	0.1%	0.3%	0.4%	0.4%	0.5%
F^-	0.13%	0.34%	2.6%	0.17%	0.42%	1.7%

4.3.2.3 Evaluation of Trust-driven Random Walks

Mohaisen et al. proposed several designs of random walks that incorporate trust between nodes in social graphs [75], and studied their impact on the performance of SybilLimit. Their findings suggest that these modified and biased random walks model trust and influence SybilLimit's performance differently. In this subsection we evaluate the performance of our sybil identification algorithm operating with these trust-driven random walks on our Facebook dataset. In the experiments the number of attack edges is fixed at 1000, $R = 2000$, and the sybil region is built with the PA model.

For lazy random walks, the transition probability form node i to node j is $\frac{1-\alpha}{d_i}$ if j is a neighbour of i, α if $i = j$, and 0 otherwise, where α is a parameter characterizing the trust level, and d_i is the degree of node i. The false rates of the sybil identification algorithm over lazy random walks ($\alpha = 0.2$) is shown in Table 4.9, which are similar to the results of simple random walks presented in Table 4.2. The reason is that by capturing the random walk in the current node with probability α, the laziness actually reduces the effective length of random walks by α, which does not degrade performance given a small α.

At each step of an originator-biased random walk, the transition probability from the current node to the originator of the random walk is α, and with probability $1-\alpha$ the next hop is chosen uniformly among the neighbours of the current

node. In the experiments we found that when operating with originator-biased random walks, our sybil identification algorithm cannot effectively identify sybil nodes. The coverage of the originator-biased random walks are severely limited by their inherent ``discontinuity'': at each step the random walk is moving back to the originator with probability α. Since our sybil identification algorithm is built upon the intuition that the coverage of random walks starting from an honest node is larger than the random walks starting from a sybil node, the originator-biased random walks do not fit our algorithm.

For two nodes i and j and their sets of neighbours N_i and N_j, the similarity between i and j is defined as $S_{ij} = \frac{N_i \cap N_j}{N_i \cup N_i}$. At each step of a similarity-based random walk, the transition probability from node i to one of its neighbours j is $\frac{S_{ij}+1}{\sum_{k \in N_i}(S_{ik}+1)}$, i.e., the more similar two adjacent nodes are, with higher possibility a similarity-based random walk will traverse the link connecting the two nodes. Table 4.9 shows the false rates of the sybil identification algorithm operating with similarity-based random walks. The results are comparable to those obtained over the weighted social network (Table 4.8). This is because like the weighted random walks are biased based on link weights, the similarity-based random walks are biased according to the similarity between each pair of adjacent nodes.

4.3.3 Evaluation of the Sybil Community Detection Algorithm

To evaluate our sybil community detection algorithm, the parameters we used in the experiments are as follows: $l_0 = 100$, $\beta = 0.95$, $R = 2000$. We test the algorithm on two social topologies, with the sybil region built through two mod-

Table 4.10: Performance of the sybil community detection algorithm (1000 attack edges)

10 sybil nodes per attack edge (10000 sybils)				
	Percentage of found sybil nodes		Number of falsely detected honest nodes	
	Orkut	Facebook	Orkut	Facebook
PA model	99.91%	99.82%	0.3	0.3
ER model	99.85%	99.84%	0	0.7
5 sybil nodes per attack edge (5000 sybils)				
	Percentage of found sybil nodes		Number of falsely detected honest nodes	
	Orkut	Facebook	Orkut	Facebook
PA model	99.86%	99.66%	0.1	0.8
ER model	99.74%	99.66%	0.1	0.2
1 sybil node per attack edge (1000 sybils)				
	Percentage of found sybil nodes		Number of falsely detected honest nodes	
	Orkut	Facebook	Orkut	Facebook
PA model	99.4%	98.4%	0.1	1.1
ER model	98.7%	98.3%	0.1	0.3

els, respectively. The number of attack edges is 1000, and the size of the sybil region depends on how many sybil nodes are introduced by each attack edge. As the goal of our sybil community detection algorithm is to detect the sybil community surrounding a known sybil node, when running each experiment we randomly select a sybil node as the starting node of our algorithm, and we get the percentage of the sybil nodes that can be detected, as well as the number of the honest nodes that are falsely detected. We repeat each experiment 20 times and calculate the mean value.

Tables 4.10 show the results when each attack edge introduces 10 sybil nodes, 5 sybil nodes, and 1 sybil node, respectively. It is easy to see that our algorithm can detect an overwhelming majority of the sybil region in all the experiments, and the numbers of falsely detected honest nodes are very small:

Table 4.11: Performance of the sybil community detection algorithm (10000 attack edges)

10 sybil nodes per attack edge (100000 sybils)				
	Percentage of found sybil nodes		Number of falsely detected honest nodes	
	Orkut	Facebook	Orkut	Facebook
PA model	99.77%	99.85%	0.2	3.7
ER model	99.90%	99.89%	0.1	3.0
5 sybil nodes per attack edge (50000 sybils)				
	Percentage of found sybil nodes		Number of falsely detected honest nodes	
	Orkut	Facebook	Orkut	Facebook
PA model	99.67%	99.69%	0.4	3.6
ER model	99.79%	99.66%	0.2	3.5
1 sybil node per attack edge (10000 sybils)				
	Percentage of found sybil nodes		Number of falsely detected honest nodes	
	Orkut	Facebook	Orkut	Facebook
PA model	99.28%	98.68%	0.4	4.3
ER model	98.88%	98.38%	0.1	3.7

Table 4.12: Performance of the sybil community detection algorithm on a weighted social network

	Percentage of found sybil nodes			Number of falsely detected honest nodes		
# of sybils	10000	5000	1000	10000	5000	1000
PA Model	99.79%	99.66%	98.6%	0.4	0.5	0.6
ER Model	99.83%	99.68%	98.4%	0.5	0.8	0.7

on average less than 1 honest node is falsely detected in each experiment. The undetected sybil nodes are all compromised nodes, the sybil nodes directly connecting to the honest nodes through attack edges. Our sybil community detection algorithm achieves high accuracy with short running time. For example, on one core of an Intel Xeon 2.93GHz processor, the time to detect a 10000-node sybil region connecting to the Facebook data set is 16 seconds, when the sybil region is constructed with the PA model, and 20 seconds when

Table 4.13: Accuracy of the Combo algorithm

	Percentage of found sybil nodes		Number of falsely detected honest nodes	
	Orkut	Facebook	Orkut	Facebook
10000 sybils	99.82%	99.64%	0	0.4
5000 sybils	99.68%	99.40%	0.2	1.0
1000 sybils	98.4%	96.5%	0.2	1.2

the sybil region is constructed with the ER model.

To investigate the scalability of our algorithm, we raise the number of attack edges to 10000 and repeat the experiments. All the parameters used in the algorithm stay the same. The results are shown in Table 4.11, which illustrates that with the size of the sybil region increasing by 10 times, our algorithm still achieves similar performance.

We also evaluated the performance of the sybil community detection algorithm on the weighted social network sample used in Section 4.3.2.3. The modified algorithm runs by performing *weighted partial* random walks. Each weighted partial random walk behaves the same as the partial random walks, i.e., it does not traverse the same node more than once. However, at each intermediate node, the next hop is chosen by considering edge weights, as described in Section 4.3.2.3. When building the sybil regions, we randomly assign weights to the edges connecting sybil nodes, such that the average weight is equal to the average weight of our data set. The results in Table 4.12 show that the number of both undetected sybil nodes and falsely detected honest nodes is very small.

4.3.4 Evaluation of the Combo Algorithm

In this subsection, we evaluate the performance of the Combo algorithm presented in Section 4.2.3. Since the Combo algorithm behaves the same as the

sybil identification algorithm when identifying sybil nodes, we measure the running time and accuracy of the Combo algorithm after a sybil node has been found and the algorithm is used to detect the sybil community surrounding the sybil node, and compare it with our sybil community detection algorithm.

In the experiments we fixed the number of attack edges at 1000, and evaluated the Combo algorithm when the number of sybil nodes in the sybil region is 10000, 5000, and 1000, respectively. The sybil region is constructed with the PA model. As mentioned in Section 4.2.3, the advantage of the Combo algorithm over the stand-alone sybil community detection algorithm is that the former reuses the simple random walks performed in the identification method, and thus avoids estimating partial random walk length and performing partial random walks, which significantly reduces computation overhead. For instance, on a single core of an Intel Xeon 2.93GHz processor, the running time for the Combo algorithm to detect a 100000-node sybil region constructed with the PA model and connecting to the Orkut graph is 91 seconds, while for the stand-alone sybil community detection algorithm it is 257 seconds, almost three times longer. Table 4.13 shows the accuracy of the Combo algorithm for sybil community detection. All the results are averaged over 20 runs. Compared with the accuracy of the sybil community detection algorithm in Table 4.10, the detection rate is slightly lower. This is because the Combo algorithm detects the sybil community by analyzing simple random walks originating from a sybil node, instead of partial random walks that are more likely to be trapped within the sybil region.

4.4 Conclusion

We present SybilDefender, a scheme that leverages the network topologies to defend against sybil attacks in large social networks. SybilDefender consists of a sybil identification algorithm, a sybil community detection algorithm, and two approaches to limiting the number of attack edges in OSNs. Our evaluation on two large-scale real-world social network samples shows that SybilDefender can correctly identify the sybil nodes, even when the number of sybil nodes introduced by each attack edge approaches the theoretically detectable lower bound, and that it can effectively detect the sybil community surrounding a sybil node with different sizes and structures.

Chapter 5

Privacy-Preserving Location Sharing in Location-based Online Social Networks

Fast Mencius and SybilDefender improve security of the infrastructure layer and the function layer of OSNs, respectively. Starting from this chapter, we investigate how to protect user privacy on OSNs. In the past few years, online social networks (OSNs) have gained great popularity and are among the most frequently visited sites on the Web [7]. Through the services provided by OSNs, users establish and strengthen connections by sharing thoughts, activities, photos, and other personal information. At the same time, the popularity of mobile devices such as cell phones and tablets is exploding, and these mobile devices are becoming smarter. Most cell phones sold today are capable of accessing the Internet over WiFi or cellular networks, and determining their location through GPS or cellular geolocation. As a result, it is not surprising to see the rapid fusion of OSNs with mobile computing, that is, a new paradigm called

location-based online social networks (LBSNs).

LBSNs can be classified into two types. The first type consists of the existing OSNs that turn mobile, like Facebook and Twitter. They tail the contents and access mechanisms for mobile users, and allow access from mobile devices. Facebook has announced that among the 1 billion active users, more than 500 million accessed Facebook through their mobile devices [6]. The second type is the newly emerging OSNs that are dedicated to mobile users, such as Foursquare. These new LBSNs are designed to explicitly take advantage of the location information provided by the mobile devices. Compared with traditional OSNs, both of these categories of LBSNs take a step further in that they provide location-based services, which are a missing link between the real world and OSNs. For example, users can use the ``friend locator'' feature provided by LBSNs to learn their friends' whereabouts. Instead of explicitly inputting their locations, the recent smartphone platforms that support various localization technologies make it much easier for the users to access and share their locations with each other.

While these location-based features make LBSNs more popular, they also raise significant privacy concerns. Users' locations may reveal sensitive private information, such as interests, habits, and health conditions, especially when they are in the hands of the adversaries. Mobile users may also be harmed by stalkers who track users with their location information [92, 93]. The threat is more serious with regard to LBSNs, because users' physical locations are now being correlated with their profile information. Considering that all the current LBSNs are under centralized control, users' location privacy will be compromised if the location data collected by the LBSNs are abused, inadvertently leaked, or under the control of hackers. Without a guarantee of privacy, users

may be hesitant to share locations through LBSNs [14, 68].

Given the popularity of LBSNs and the sensitivity of the location data users place in them, it is critical to limit the privacy risks posed by today's LBSNs while retaining their functionality. As indicated in previous research [62], location and presence are two sources of privacy leakage introduced by LBSNs. Smoke-Screen [32] considers the problem of how to flexibly share presence with both friends and strangers while preserving user privacy. However, until now no scheme has been proposed to address the same problem for location sharing in LBSNs. Previous work [85, 115] discussed sharing locations between established relations in a privacy-preserving way. Nevertheless, restricting location sharing to established social relations makes a large class of location-based social applications, such as Serendipity [36], impossible. In an LBSN, users may want to see the locations of both friends and strangers within some range, while at the same time they should be able to control how their own location information is accessed by others. To protect users' location privacy, the system should work in such a way that an adversary controlling the LBSN cannot obtain users' location information. Unfortunately, no scheme proposed so far meets these requirements.

In this chapter, we present a privacy management system called MobiShare, which provides flexible privacy-preserving location sharing in LBSNs. Our system is *flexible* in that it supports the features of location sharing in real-world LBSNs, including sharing locations between both trusted social relations and untrusted strangers, querying locations within a certain range, and user-defined access control. MobiShare leverages the existing OSNs and requires no change to their architectures, but these OSNs are not trusted to access users' location information. In MobiShare, the social network server stores users' identity-

related information, while an untrusted third-party location server stores users' anonymized location updates that are mixed with dummy location updates. The adversary cannot link a precise location to an identified user, unless he controls both entities. A user's location information is not leaked to the malicious users who are unauthorized to access his locations either, even if these malicious users collude with the social network server or the location server. To investigate the feasibility of MobiShare, we have implemented an experimental system. The evaluation results show that MobiShare only consumes a very limited amount of the battery power and system resources of the mobile devices, and the deployment overhead it imposes on the cellular towers is small.

The rest of this chapter is organized as follows: in Section 5.1, we describe MobiShare's system architecture, trust and threat model, and system goals. In Section 5.2, we present the design of MobiShare. The security analysis of MobiShare is presented in Section 5.3. The feasibility of MobiShare is shown experimentally in Section 5.4, and in Section 5.5 we conclude the chapter.

5.1 System Architecture and Threat Model

5.1.1 System Model

To protect users' location privacy, MobiShare stores users' identity-related information and anonymized location updates at two separate entities, the social network server and the location server. Figure 5.1 shows the system architecture of MobiShare, which consists of four components: the social network server, the location server, the cellular towers, and the clients. The social network server can be a server of any existing OSN that would like to implement

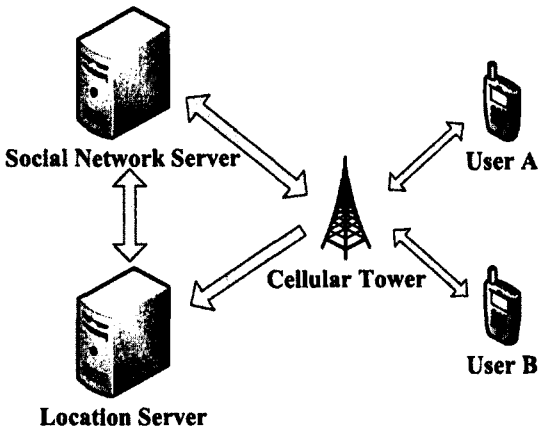


Figure 5.1: System architecture

the location-sharing service. The motivation for an OSN to provide a privacy-preserving service is to alleviate users' concerns about the abuse of their private information, and convince them to share their locations. Like today's OSNs, the social network server manages users' identity-related information, e.g., their profiles and friend lists. The location server is an untrusted third-party server that stores the users' anonymized location updates. For example, a company may implement the location server so as to profit from the OSNs or the users. Also, some privacy advocacy organization, like the Electronic Frontier Foundation (EFF), may provide the location server to help protect user privacy. The tasks of the location server can also be offloaded to cloud computing providers such as Amazon EC2 [2]. Given that all the current smartphones are equipped with the function to access the Internet with wireless techniques like 3G/LTE, it is reasonable to assume that users can communicate with the servers through cellular networks.

We assume that each user has a unique identifier at the social network server. For instance, on Facebook the user identifier is the large integer shown in the URL of each user's profile page. This identifier is used as his identity in

MobiShare. Each user generates by himself a public-private key pair and a symmetric session key, and shares the session key with all his social network friends. Each cellular tower has a unique identifier and generates by itself a symmetric secret key, and shares them with the location server. The secret key is used to prevent the social network server from prying the contents of the replies sent form the location server. The location server also shares one symmetric secret key with all the cellular towers. The servers and the cellular towers are connected by high-speed secure links, and the social network server cannot identify the communicating cellular towers by observing the IP addresses in the connections. This can be achieved by using proxies provided by the cellular carriers.

5.1.2 Trust and Threat Model

The social network server and the location server are not trusted to access users' location information. We assume that either the social network server or the location server can be compromised and controlled by an adversary seeking to link users' identities to their locations, but the adversary cannot control both entities. This model is rational in that many security breach cases usually involve the hack of databases or logs in a single system, or dishonest insiders within a system trying to fetch sensitive information [4]. It is unlikely that two servers operated by independent organizations can be controlled by the same adversary. In addition, some users may also be malicious and seek to obtain the location information they are unauthorized to access. The social network server or the location server may collude with these malicious users. For example, an employee of the social network company may register for the location-sharing

service, and collude with the server to extract other users' location information.

This work does not investigate how to improve location privacy within the cellular networks. The wireless Enhanced 9-1-1 rules [5] of the Federal Communications Commission (FCC) require that the cellular carriers can locate the subscribed cell phones with an accuracy of 50 to 300 meters, depending on the type of technology used. Also, for each subscribed cell phone the cellular carrier generally knows the owner's name and address. Therefore, we make no attempt to conceal the devices' locations from the cellular networks, i.e., *the cellular towers are trusted.* In MobiShare, location anonymization is done by the cellular towers. To avoid interfering with the real-time and prioritized telephony services on cellular towers, the functions required by MobiShare can also be offloaded to trusted machines connecting the cellular towers, which are administrated by cellular carriers.

5.1.3 System Goals

MobiShare is designed to achieve the following goals.

Privacy. When using the service, users' location information should not be leaked to the social network server, the location server, or unauthorized users, even if the social network server or the location server colludes with malicious users.

Flexible. Our design should support the features of location sharing in real-world LBSNs, which means that the users should be able to share locations between both trusted social relations and untrusted strangers, query the locations of other users within a certain range, and retain control over how their own location information is accessed by others.

Table 5.1: Summary of notations

ID_A	User A's social network identifier, used as his identity
FID_A	User A's fake ID
$PubKey_A$	User A's public key
$PrivKey_A$	User A's private key
$SessKey_A$	User A's session key, shared with all his friends
df_A	User A's friend-case threshold distance
ds_A	User A's stranger-case threshold distance
CID_C	Cellular tower C's identifier
$SecKey_C$	Cellular tower C's secret key, shared with the location server
$SecKey_{Loc}$	Location server's secret key, shared with the cellular towers

Mobile device friendly. Considering the limited resources on the mobile devices, our client should incur small computation and storage overhead, as well as low power consumption.

5.2 System Design

In this section we describe how we achieve our design goals. We separate the problem of privacy-preserving location sharing into two cases, sharing locations between friends and between strangers, and solve them separately. The intuition behind our design is that the users' identity-related information and anonymized location updates are stored and processed at two separate entities, and no single entity can accumulate enough information to breach users' location privacy. MobiShare's messaging protocol consists of the following components: (1) registering for the location-sharing service; (2) establishing an authenticated communication link between a user and a cellular tower; (3) up-

dating locations; (4) querying friends' locations within a certain range; (5) querying nearby strangers' locations within a certain range. They will be described in detail in the following subsections. A summary of the notations used is given in Table 5.1.

5.2.1 Service Registration

Before using the location-sharing service, each user needs to register for the service at the social network server. During registration, user A shares his public key $PubKey_A$ with the social network server, and defines his access control settings, which consist of two threshold distances, df_A and ds_A. df_A is the threshold distance within which A is willing to share his location with his social network friends. If the distances between A and some of his friends are larger than df_A, they cannot access A's current location. Similarly, ds_A is the threshold distance within which A agrees to share his location with arbitrary users. In our implementation, users can select their threshold distances from two drop-down lists. If a user does not want to share his location, he can simply choose 0 as the threshold distance. After registration, A keeps a record of his social network identifier ID_A, while the social network server stores an entry $\langle ID_A, PubKey_A, df_A, ds_A \rangle$ in its *subscriber* table, where the user identity is the primary key.

5.2.2 Authentication

Figure 5.2 shows the messages involved in the authentication process. After user A's handset connects to cellular tower C, an encrypted data transmission link is established based on mobile telecommunication techniques such

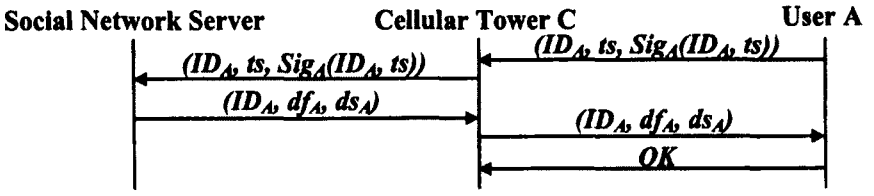


Figure 5.2: Authentication

as 3G/LTE. To let the cellular tower authenticate his identity, A sends an authentication request $(ID_A, ts, Sig_A(ID_A, ts))$ to the cellular tower, where ID_A is A's social network identifier, and ts is a timestamp used to prevent replay attack. The message is signed by A's private key. The cellular tower forwards this message to the social network server. Upon receiving the message, the social network server searches its subscriber table for A's registration information, including A's public key $PubKey_A$ and the threshold distances df_A and ds_A.

The social network server first uses $PubKey_A$ to verify A's signature. If the verification succeeds, it sends a reply (ID_A, df_A, ds_A) to the cellular tower. The cellular tower forwards this message to A. A checks if ID_A, df_A, and ds_A are correct. If so, A sends an OK message to the cellular tower. On the reception of the OK message, the cellular tower stores an entry $\langle ID_A, df_A, ds_A \rangle$ in its *user info* table, where the user identity is the primary key. After this an authenticated communication link is established between A and the cellular tower, and A's identity is attached to this link.

5.2.3 Location Updates

When the users upload their location updates, the task of the cellular towers is to perform anonymization such that the user identities cannot be inferred from the anonymized location updates. This is achieved by leveraging both pseudonyms

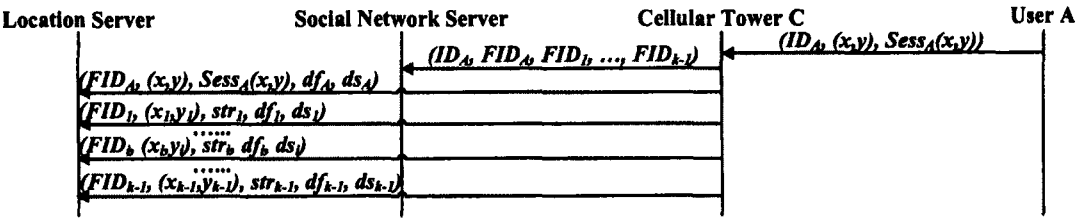


Figure 5.3: Location update

and dummy location updates. We assume that each cellular tower periodically generates fake IDs, and saves them in a fake ID pool. The size of the pool depends on the number of users connecting to this cellular tower and their location update frequency. Each location update from a user consumes k fake IDs. The fake IDs can be efficiently generated using a cryptographic hash function and a random salt value as follows: $fake\ ID_i = SHA(fake\ ID_{i-1} \oplus salt)$.

Figure 5.3 shows the messages involved in the location update process. User A periodically gets his current location through techniques such as GPS or cellular geolocation. To update his location, A sends a message $(ID_A, (x, y), Sess_A(x, y))$ to the cellular tower, where (x, y) is A's current location, and $Sess_A$ (x, y) is the location encrypted with A's session key. This session key is shared with all his social network friends. Upon receiving the location update from A, the cellular tower performs coarse location verification by checking if (x, y) is within its working range. If so, the cellular tower keeps a record of A's current location in its user info table. Then the cellular tower picks k fake IDs from its fake ID pool. One of the k fake IDs is used to replace A's identity in the real location update. Let this fake ID be FID_A. The other $k - 1$ fake IDs are used by the cellular tower to construct dummy location updates. The cellular tower stores FID_A at A's entry in the user info table, and sends the mapping between A's identity and the k fake IDs to the social network server, which stores an entry

$\langle ID_A, FID_A, FID_1, ..., FID_{k-1} \rangle$ in its *fake ID* table, where the user identity is the primary key. With this table the social network server knows, given any user identity, the fake ID used in the latest real location update and the fake IDs used in the latest dummy location updates.

To anonymize the location update from A, the cellular tower sends k location updates to the location server. Only one location update contains A's real location, while the other $k-1$ are dummies. The real location update is of the form $(FID_A, (x,y), Sess_A(x,y), df_A, ds_A)$. It consists of A's fake ID, plaintext and encrypted locations, and the threshold distances. To construct the dummy location updates, the cellular tower follows the method proposed by Kido et al. [58] and generates $k-1$ dummy locations within its coverage. The i^{th} dummy location update is of the form $(FID_i, (x_i, y_i), str_i, df_i, ds_i)$. FID_i is one of the k fake IDs extracted from the fake ID pool; (x_i, y_i) is one of the dummy locations generated by the cellular tower; str_i is a random string imitating the encrypted location; df_i and ds_i are the threshold distances of a random user whose information is stored in the cellular tower's user info table. Note that for the dummy location updates, the cellular tower does not need to encrypt the locations. It only needs to generate arbitrary strings with the length of an encrypted location. Like generating fake IDs, these strings can be created efficiently using a hash function and a salt value.

The cellular tower sends the k location updates to the location server in a random order with random temporal intervals following the exponential distribution. The location server stores them in its *location update* database. The database consists of a number of tables. Each table represents a geographic region. Updates of locations within a region are stored in the corresponding table, where the fake ID is the primary key. Organizing the location update database in this

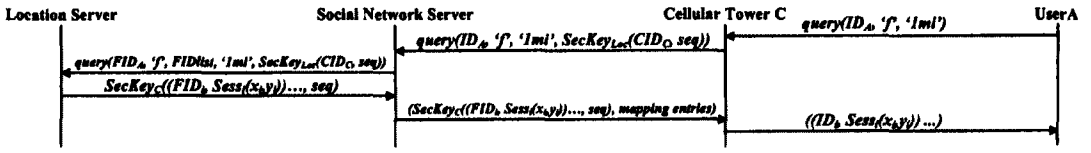


Figure 5.4: Querying friends' locations

way improves search efficiency. For instance, given one location, to find all the fake IDs within a range, instead of checking all the stored location updates, the location server only needs to check the tables of the regions that overlap the queried circular area. The entries in the location update database expire after a certain period of time.

The rationale behind this design is that the users' location updates are separated into two parts: user identities and locations. The mappings between user identities and pseudonyms are presented to the social network server, while the anonymized location updates are sent to the location server. No single entity is able to link users' identities to their locations.

5.2.4 Querying Friends' Locations

Figure 5.4 shows the messages involved in querying friends' locations. To query the locations of his friends within a certain range, say 1 mile, user A sends *query*(ID_A, *'f'*, *'1mi'*) to the cellular tower. The cellular tower appends its identifier and a sequence number to this message, which are encrypted by the location server's shared secret key, and forwards *query*(ID_A, *'f'*, *'1mi'*, $SecKey_{Loc}(CID_C, seq)$) to the social network server. The cellular identifier will be used by the location server to find the secret key shared by this cellular tower so as to encrypt the reply. The sequence number is added such that the social network server cannot infer that two queries of different users come from the

same cellular tower, since for the same CID_C, $SecKey_{Loc}(CID_C, seq)$ veries with seq.

Upon receiving the query, the social network server looks up the currently used fake IDs of A and all A's friends in its fake ID table. Let $FIDlist$ be a list consisting of the fake IDs of all A's friends in random order, including the fake IDs used in each friend's latest real location update and the fake IDs used in each friend's latest dummy location updates. Assume A has f friends, then the size of $FIDlist$ is kf. The social network server replaces A's identity with FID_A, which is the fake ID used in A's latest real location update, and sends *query*(FID_A, *'f', FIDlist, '1mi'*, $SecKey_{Loc}(CID_C, seq)$) to the location server. On the reception of the query, the location server checks which fake IDs in $FIDlist$ are within 1 mile of FID_A. For each of these nearby fake IDs, the location server enforces access control based on that fake ID's friend-case threshold distance stored in the location update database. For example, if one fake ID is 0.7 miles away from FID_A, but its friend-case threshold distance is 0.5 miles, the location server will not send the location of this fake ID to A, even though it is within the queried range.

After finishing distance computation and access control enforcement, the location server sends a reply $SecKey_C((FID_i, Sess_i(x_i, y_i))..., seq)$ to the social network server. To prevent the social network server from prying the contents, this message is encrypted with the cellular tower's shared secret key. The reply may contain multiple location entries. Each entry is of the form $(FID_i, Sess_i(x_i, y_i))$, where FID_i is a fake ID in $FIDlist$ that is within the queried range and has passed access control enforcement, and $Sess_i(x_i, y_i)$ is FID_i's encrypted location stored in the location update database. As mentioned in the previous subsection, if FID_i is a fake ID used in the latest real location update of a us-

er, then $Sess_i(x_i, y_i)$ is FID_i's location encrypted with this user's session key shared with all his friends, including A. Otherwise, if FID_i is a fake ID used in one of the dummy location updates, then $Sess_i(x_i, y_i)$ is an arbitrary string with the length of an encrypted location. To process the friend-case query, the location server uses the stored plaintext locations to compute distances, while it sends the encrypted locations back to the querier. The purpose of this design is to defend against the social network server colluding with malicious users, which will be explained in Section 5.3.

Upon receiving the reply, the social network server appends a mapping entry for each of A's friends to the message, and forwards the reply to the cellular tower. Each mapping entry is of the form (FID_j, ID_j), where FID_j is the fake ID used in friend j's latest real location update, and ID_j is friend j's identity. Note that the social network server does not provide the mapping entries for the fake IDs used in the dummy location updates. Assume A has f friends, then f mapping entries are appended to the reply. The reply received by the cellular tower consists of the encrypted location entries and the mapping entries. The cellular tower first uses its secret key shared with the location server to decrypt the location entries. Then for each location entry that has a matching mapping entry with the same fake ID, it replaces the fake ID in the location entry with the user identity. The location entries that do not have a matching mapping entry, which are from the dummy location updates, are all discarded by the cellular tower. Until now each remaining location entry has the form $(ID_i, Sess_i(x_i, y_i))$, which includes both the user identity and the encrypted location. The cellular tower sends these entries to A. Since we assume that all of A's friends have shared their session keys with A, A can decrypt and get the plaintext locations of the nearby friends.

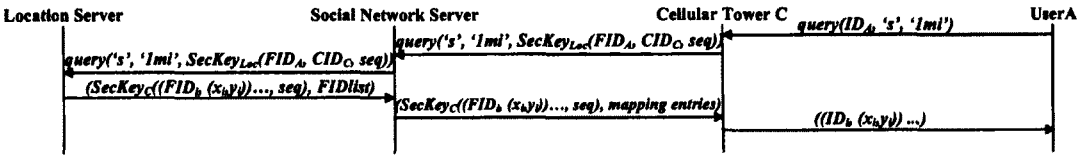


Figure 5.5: Querying strangers' locations

5.2.5 Querying Strangers' Locations

Figure 5.5 shows the messages involved in querying strangers' locations. To query the locations of arbitrary users within a certain range, say 1 mile, A sends *query*(ID_A, *'s'*, *'1mi'*) to the cellular tower. The cellular tower keeps a record of the queried range at A's entry in the user info table. Then it removes ID_A, and appends FID_A, which is the fake ID used in A's latest real location update, the cellular tower identifier, and a sequence number to the message, all of which are encrypted by the location server's secret key. The cellular tower sends *query*(*'s'*, *'1mi'*, $SecKey_{Loc}(FID_A, CID_C, seq)$) to the social network server, which directly forwards the query to the location server.

On the reception of the query, the location server looks up the fake IDs that are within 1 mile away from FID_A. For each of these fake IDs, the location server enforces access control based on its stranger-case threshold distance. Assuming there are n nearby fake IDs that pass access control enforcement, the location server randomly picks another recently received $(k-1)n$ fake IDs from the location update database. These fake IDs are mixed with the n nearby fake IDs in the reply to achieve k-anonymity. The reply sent from the location server to the social network server is of the form $(SecKey_C((FID_i, (x_i, y_i))..., seq), FID\ list)$. This message includes n location entries, each of which contains a nearby fake ID and its plaintext location. All these location entries are encrypted with the cellular tower's secret key. $FIDlist$ consists of the n nearby fake IDs mixed

with the $(k-1)n$ randomly selected fake IDs. On the reception of the reply, the social network server cannot pry the contents of the encrypted location entries, nor can it learn which fake IDs in $FIDlist$ are currently close to A, since it cannot distinguish the n nearby fake IDs from the $(k-1)n$ padded fake IDs.

As mentioned in Section 5.2.3, to anonymize each location update from a user, the cellular tower generates $k-1$ dummy location updates and sends all the k updates to the location server. Therefore, approximately $(k-1)/k$ of the kn fake IDs in $FIDlist$ come from dummy location updates. The social network server filters out all these fake IDs based on its fake ID table. For each of the remaining fake IDs, which are the fake IDs that have been used in the real location updates, the social network server appends to the reply a mapping entry (FID_j, ID_j, ds_j), where ID_j is user j's identity, and ds_j is user j's stranger-case threshold distance. Then it sends $(SecKey_C((FID_i, (x_i, y_i))..., seq), mapping\ entries)$ to the cellular tower.

Upon receiving the reply, the cellular tower first uses its secret key to decrypt the location entries. To defend against the location server colluding with a malicious user, the cellular tower randomly selects one location entry that has a matching mapping entry in the reply, and checks if the distance between the location in this entry and A's current location stored in the user info table is smaller than both the queried range and the stranger-case threshold distance in the mapping entry. If the check fails, the reply is discarded and the location server is suspected of behaving maliciously. Otherwise, for each location entry that has a matching mapping entry, the cellular tower replaces the fake ID with the user identity. The resulting location entry is $(ID_i, (x_i, y_i))$. The cellular tower sends all these entries to A. Until now A learns both the identities and locations of the nearby users who are willing to share their whereabouts.

The maximum stranger-case query range is limited in MobiShare. For example, in our implementation, a user is allowed to query the locations of arbitrary users within at most 2 miles. Without such limit, a malicious user can launch a DoS attack on the location server by querying the locations of strangers within a very large range.

5.3 Security Analysis

In this section we present the intuition behind the privacy guarantee provided by MobiShare. We discuss the security of MobiShare after an adversary compromises either the social network server or the location server, or controls some malicious users. We also consider the collusion between either of the servers and the malicious users.

Compromised social network server. The adversary succeeds in breaching a user's location privacy if he can associate the user's identity with a location. By controlling the social network server, the adversary can access all the records in the subscriber table and the fake ID table, as well as observe and tamper with the users' queries and the corresponding replies.

Entries in the subscriber table are of the form $\langle ID_A, PubKey_A, df_A, ds_A \rangle$, and entries in the fake ID table are of the form $\langle ID_A, FID_A, FID_1, ..., FID_{k-1} \rangle$. It is easy to see that these two tables only store users' identity-related information. By accessing them the adversary cannot obtain any location information. The adversary may observe users' queries. However, the friend-case query only contains the querier's identity in plaintext. Since we assume that the social network server cannot identify the communicating cellular towers by observing the IP addresses in the connections, it cannot infer which cellular tower the querier

currently connects to. By observing the stranger-case queries the adversary learns even less, because the querier's fake ID is encrypted. The adversary may observe the replies sent from the location server. However, in both the friend case and the stranger case, the location entries in a reply are encrypted by the secret key of the cellular tower from which the query was sent, and the adversary cannot pry the contents.

The adversary may compromise the social network server and make it not follow our protocol. For example, upon receiving a friend-case query the social network server may replace the querier's identity with an incorrect fake ID, or it may append incorrect mapping entries to a reply sent from the location server. These disruptive behaviors, however, do not help the adversary to learn users' locations, and they significantly increase the risk of being detected.

Compromised location server. If the adversary controls the location server, he can access all the stored location updates. However, these location updates are all anonymized, from which the adversary cannot infer user identities. In our design, pseudonyms are used to replace users' identities in the location updates. For each location update from a user, $k - 1$ dummy location updates are generated and uploaded to the location server. This makes the adversary unable to distinguish the real location data so as to identify the users.

Malicious users. Malicious users may seek to fetch other users' location information they are unauthorized to access. In MobiShare, to learn the locations of others, a user simply sends out a query and waits for the reply, and he has no control over how the query is processed. A user cannot get the location of another if the distance between them is larger than the queried range or the threshold distance defined by the latter. Therefore, the only way for a malicious user to fetch the location information he should not know is to falsify his current

location. However, this is forbidden in our design, because the cellular towers perform location verification when they receive location updates from the users. If the verification fails, the location update will be discarded and the sender will be suspected. Our current design requires that the cellular towers perform coarse verification. That is, each cellular tower checks if the reported location is within its working range. If this is not enough, the cellular towers can also perform precise verification, by comparing the updated location with the location of the sender's mobile device known by the cellular network. Only if the difference is smaller than a threshold will the update be accepted.

Collusion between the social network server and malicious users. The social network server and some malicious users may collude to compromise other users' location privacy. Since in our design the problem of privacy-preserving location sharing has been separated into two cases, sharing between friends and between strangers, we will discuss our privacy guarantee under this threat in both cases.

The social network server controlled by the adversary can distinguish the colluding users' friend-case queries from the other queries by observing the user identity field in the received queries. On the reception of a friend-case query from a colluding user, the social network server can put arbitrary users' fake IDs into the fake ID list sent to the location server, instead of the fake IDs of the querier's friends. In this way, the reply from the location server may contain some of these users' locations, depending on the distance between each user and the querier. However, neither the social network server nor the malicious user can see the locations, because each of these locations in the reply is encrypted by the corresponding user's session key shared only with his friends.

On the other hand, upon receiving a stranger-case query from a colluding user, the social network server can do nothing, because the cellular tower already replaces the querier's identity with the fake ID used in his latest real location update, and encrypts the fake ID with the location server's secret key. When the social network server receives the reply sent from the location server, it does not help the colluding user if the social network server appends arbitrary users' mapping entries to the reply, since the reply only contains the location entries of the nearby users.

Collusion between the location server and malicious users. The location server and the malicious users controlled by the same adversary may collude to breach user privacy. Again, we will discuss this threat in both the friend case and the stranger case.

The location server controlled by the adversary can identify the queries from colluding users by matching the location of the querier with the location of a malicious user. Upon receiving a friend-case query from a colluding user, the location server may put the encrypted locations of all the friends of the colluding user into the reply, without calculating distances or enforcing access control. In this way, the colluding user can learn the current locations of all his friends, even if some friends are outside the queried range or their locations should not be shared with the colluding user based on their friend-case threshold distances. However, we assume that in an LBSN users are cautious when establishing new friendships, especially when they know that their friends can see their whereabouts. Thus, it is unlikely that a malicious user can make friends with many honest users, and the location privacy breach is very limited. Even if this assumption does not hold in some LBSNs, the attack can be defended against by letting the cellular towers sign both the plaintext and encrypted locations when

they receive a location update. The signature guarantees that the plaintext location is linked to the encrypted location, and they cannot be modified. When the location server sends back the reply, each entry contains the signed plaintext and encrypted locations. The social network server appends each friend's friend-case threshold distance to the reply. After the cellular tower receives the reply, it randomly selects one location entry and checks its validity, by verifying the signature and examining if the distance between the plaintext location and the location of the querier is smaller than both the queried range and the threshold distance. The reply is discarded if the check fails.

On the reception of a stranger-case query from a colluding user, the location server controlled by the adversary may put arbitrary fake IDs' locations into the reply, for the purpose that their identities can be learned by the colluding user when he gets the reply. However, as described in Section 5.2.5, before the colluding user receives the reply, the cellular tower checks the validity of the returned location entries, based on the colluding user's location and the stranger-case threshold distance of the fake ID in the randomly selected location entry, so this attack can be easily detected.

5.4 Evaluation

In this section we first describe the implementation of our system, and then we present the experimental results.

5.4.1 System Implementation

We have implemented an experimental system based on the design presented in Section 5.2. Our system consists of four components: the client, the cellular tower, the social network server, and the location server. The client is implemented in JAVA on top of the Android 2.2 platform. We run the client on a MOTOROLA DROID 2 Global smartphone, which is equipped with an ARM7 1.2GHz processor and 512MB RAM. The device can access the Internet and communicate with the laptop emulating the cellular tower through Verizon's 3G data service or WLAN, and get its current location with network-based triangulation, GPS, or Assisted GPS (A-GPS). Among them A-GPS is the preferred service, as it consumes less battery power and provides enhanced localization performance, compared with standalone GPS.

In the experiments we set up a laptop to emulate the cellular tower. A service written in Java running on the laptop processes the location updates and queries from the clients, as well as the replies from the social network server. The laptop communicates with the two servers through our campus's wireless network. In MobiShare, the social network server and the location server are supposed to be managed by two separate organizations, so we cannot assume that they are connected by local-area networks. To evaluate the performance of MobiShare in wide-area networks, in our implementation we deploy the social network server and the location server on two third-party cloud hosting services provided by JoyentCloud and Linode, respectively. Both servers are written in Java and use MySQL 5.0 to manage their local tables. The servers maintain two copies for each table. The in-memory heap tables are added to speed up search operations.

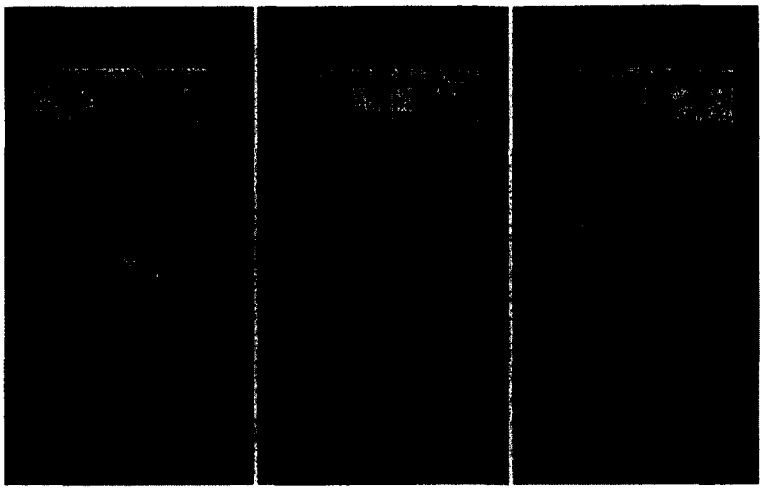


Figure 5.6: Client interface

In a separate research project we implemented and deployed a Facebook application [103]. Through this application we collected the friend lists of 386 Facebook users with their permission. The resulting data set consists of the Facebook identifiers of 48,014 users, as well as the social network topology among them. We use this data set as a social network sample in our experiments, and store it at the social network server. For each of these users we generate a location update, which includes a fake ID, a random location within 20 miles of our department building, the location encrypted by a symmetric key, and randomly selected threshold distances. All the location updates are stored at the location server. In the experiments, an author with 132 friends in our data set used the smartphone to update his location and send queries. We set k, the anonymity level, to be 5, and we use 128-bit AES for symmetric key encryption and decryption.

The size of our client executable is 252KB, which is almost negligible compared with the 8GB storage space of our smartphone. When running, the client has a memory footprint of 12MB, 2.3% of the smartphone's main memory. Fig-

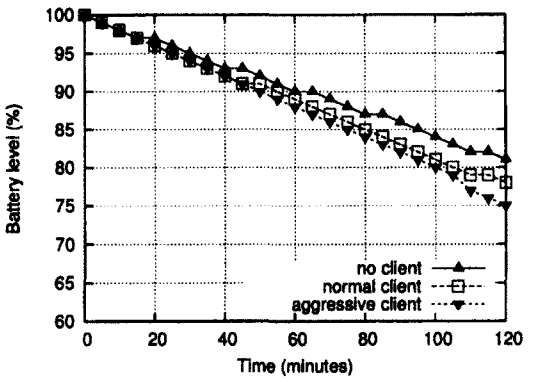


Figure 5.7: Power consumption of the client

ure 5.6 shows the client interface. It consists of three pages. The Control page allows the user to start and stop using the location-sharing service, find friends or strangers within certain ranges, and define the friend-case and stranger-case threshold distances. The Users page lists the identities of the nearby friends or strangers, depending on the type of the query. The Map page shows the locations of the nearby friends or strangers on Google Map. Each user is represented as a needle. By tapping on a needle the corresponding user's identity is shown as a toast notification on the screen.

5.4.2 Experimental Results

Power consumption of the client. We run experiments on the smartphone to investigate the power consumption of our client. When the client is running, we measure the battery level of the smartphone every 5 minutes during a period of 2 hours. In comparison, when the client is not running, we measure the battery level every 5 minutes during the same length of time. In all the experiments, we start with a fully charged battery after charging for the same amount of time. The screen and the 3G module of the smartphone are always kept on. To estimate

the power consumption of the client in the real environment, we let the client communicate with the laptop emulating the cellular tower through Verizon's 3G data service, and get its current location with A-GPS.

In MobiShare, each client periodically updates its current location, and occasionally queries the locations of friends or nearby strangers. To investigate how the location update frequency and the query frequency influence power consumption, in the experiments we define a ``normal" client as a client that updates its location every 1 minute, and queries the locations of friends or nearby strangers every 5 minutes. We also define an ``aggressive" client as a client that updates its location every 30 seconds, and queries the locations of friends or nearby strangers every 1 minute. Figure 5.7 compares the power consumption when the client is not running with the power consumption when the normal client or the aggressive client is running. As shown in the figure, when the client is not running, the battery level of the smartphone drops to 81% after 2 hours with the screen and the 3G module kept on. In comparison, when the normal client is running, the battery level drops slightly faster. After 2 hours the remaining battery level is 78%. This means that each hour our client only consumes 1.5% of the battery power. Considering that the battery capacity of our smartphone is 1390 mAh, this corresponds to 20.85 mAh per hour. When the aggressive client is running, the battery level drops to 75% after 2 hours, which indicates that each hour the more frequent location updates and queries consume another 1.5% of the battery power. The results show that the power consumption of our client is small, and the location update frequency and the query frequency have a limited impact on the battery life.

CPU utilization of the client. We also measure the CPU utilization of the client. During 2 hours of continuous running, the normal client consumes 24

seconds of CPU time, with average CPU utilization of 0.3%, while the aggressive client consumes 38 seconds of CPU time, with average CPU utilization of 0.5%. This shows that the computation overhead incurred by the client is small, since in MobiShare the tasks of clients do not involve intensive computations.

Response time. We measure the temporal interval between the time when the client sends out a friend-case query, whose queried range is 5 miles, and the time when it receives the reply. We repeat the experiment 50 times and compute the mean and standard deviation of the response time. The average response time is 386 milliseconds, with a standard deviation of 56 milliseconds. Users can hardly sense the delay. In the experiments we observed that the response time is influenced by the traffic on the wide-area networks connecting the servers and the cellular towers.

Deployment overhead on the cellular towers. MobiShare requires that the cellular towers take part in the messaging protocol. The cellular carriers can recover their deployment cost by charging the LBSNs or the users. To prevent interfering with existing services, however, the additional overhead imposed by MobiShare on the cellular towers must be small. In the experiments we use a laptop to emulate the cellular tower, which is equipped with an Intel Core Duo T8100 2.1GHz processor and 2 GB main memory. To investigate the overhead incurred by our scheme, we create a large number of dummy users on another laptop, and connect those users to the emulated cellular tower. Each user updates a randomly generated location every 1 minute, and sends a query, either friend-case or stranger-case, every 5 minutes.

Figure 5.8 shows the average CPU utilization of the cellular tower service with different numbers of connecting users. Figure 5.9 shows the corresponding memory usage. When there are 1000 users, the cellular tower service only

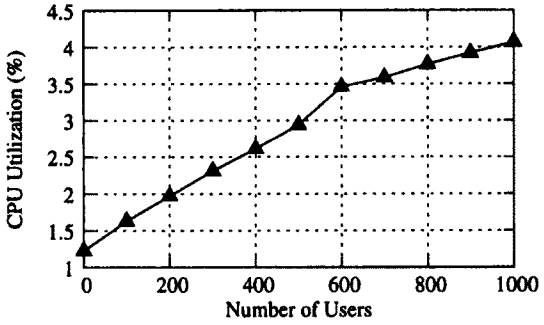


Figure 5.8: CPU utilization of the cellular tower service

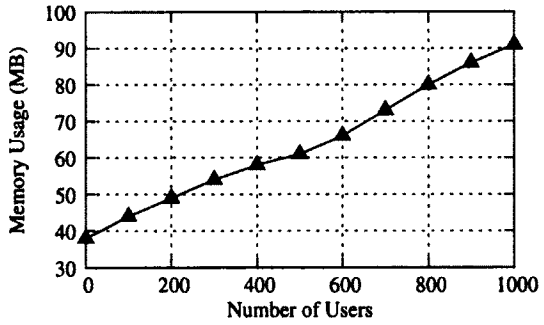


Figure 5.9: Memory usage of the cellular tower service

uses 4.1% of the CPU power and 91MB memory. The results demonstrate that our scheme consumes a very limited amount of system resources on cellular towers.

5.5 Conclusion

In this chapter, we present MobiShare, a privacy management system that provides flexible privacy-preserving location sharing in LBSNs. MobiShare supports sharing locations between both trusted social relations and untrusted strangers. With MobiShare, users can find their friends or strangers within a certain range, and control how their location information is accessed by others. By separating user identities and anonymized location updates at two entities,

users' location privacy is protected if either entity is compromised by the adversary. Our evaluation on an experimental system shows that MobiShare is mobile-device friendly, and it imposes a small additional overhead on the cellular towers.

Chapter 6

Modeling Location-based Online Social Networks

Besides hiding the location information from LBSNs, as we have done in MobiShare, another way to preserve users' location privacy on LBSNs is to protect the location data collected by LBSNs from being leaked to adversaries. Users of LBSNs can *check-in* at different venues (e.g., airports, restaurants) and notify their friends, sharing with friends information about the places they visited. These check-ins, combined with the online friendship connections revealed through the LBSNs, provide an unprecedented opportunity to study human socio-spatial behaviors based on large-scale voluntarily contributed data. This in turn facilitates a variety of services, such as urban planning, friendship recommendation, place of interest recommendation, traffic forecasting, marketing campaigns, and epidemiological modeling.

However, it is difficult to perform direct measurements of existing LBSNs, which usually take approaches to defend against automated crawlers. For example, Foursquare, the most popular LBSN, requires user authorization to col-

lect personal information, and it has limited the access rate. As a result, a direct measurement typically incurs high time and resource costs [27, 28]. To circumvent this difficulty, researchers have resorted to the publicly available datasets. Nevertheless, the number of LBSN datasets available to the community is very limited. This is mainly due to the concerns of compromising user privacy and the high costs of distributing large datasets. Users' locations may reveal highly sensitive private information, such as interests, habits, and health conditions, especially when they are in the hands of adversaries. The threat is more serious with regard to LBSNs, because users' physical locations are now being correlated with their profile information. Even if the datasets are anonymized before being published, users' identities can still be recovered from the anonymized location traces and social graphs [63, 76]. Therefore, these privacy concerns strongly discourage sharing LBSN datasets. Given the soaring adoption of LBSNs, the lack of available datasets has significantly impeded the research in this area.

An attractive alternative to shared original datasets is the synthetic datasets generated by measurement-calibrated models. There are three advantages of using synthetic datasets as replacements for real datasets. First, the synthetic datasets are randomly generated, and thus they do not compromise any user privacy. Second, compared with sharing the large datasets, the cost of sharing the models is negligible. Third, LBSN datasets with different properties can be generated on demand, which can help researchers improve the statistical confidence in their experimental results. Previous work investigated the graph models that produce synthetic social graphs of online social networks [54, 59, 88, 89]. Given all these advantages of the model-generated LBSN datasets, however, no LBSN model has been proposed in literature.

In this chapter, we propose LBSNSim, a trace-driven model for generating synthetic LBSN datasets that capture the characteristics of the real datasets. We first analyze the data from three LBSNs: Foursquare, Gowalla, and Brightkite. Our findings suggest that the LBSNs share many universal social and spatial properties. For example, the user check-in numbers follow an exponentially truncated power law distribution. The displacements between consecutive check-ins made by each user follow a two-segment distribution, whose transition point has a clear meaning. Similarly, the temporal intervals between consecutive check-ins also follow a two-segment distribution. Additionally, the friend distances follow a truncated Weibull distribution. Previous work only show that some measurements of LBSNs, such as the check-in numbers and displacements, exhibit a heavy-tail pattern [27, 81]. To the best of our knowledge, this is the first time that specific distributions have been found and explained for a wide range of statistical features of LBSNs.

Based on our findings we develop LBSNSim, which takes as input a set of known venues and model parameters, and outputs the check-in history of all the synthetic users and their friendship graph. Our model consists of three components: generating the initial location of each user, building the friendship graph by considering both social and spatial factors, and generating all the check-ins of each user.

We evaluate the fidelity of LBSNSim by comparing the properties of the real LBSN datasets with their synthetic model-generated counterparts. The results demonstrate that LBSNSim provides an accurate representation of the target LBSNs: it generates synthetic datasets that accurately capture the statistical features of the original datasets. Besides, our application-level test shows that the application results obtained by using the model-generated datasets closely

Table 6.1: Statistics of the datasets

Dataset	Users	Edges	Check-ins	Venues
Gowalla	196,591	950,327	3,674,591	675,483
Brightkite	58,228	214,078	2,920,919	476,744
Foursquare	93,115	NA	7,956,679	428,343

match those obtained by using the original datasets, which validates the feasibility of substituting real datasets with synthetic datasets. As the first generative model of LBSNs, LBSNSim has wide applications for the research community and in guiding the design of the systems and applications centered on LBSNs.

The rest of this chapter is organized as follows: Section 6.1 describes the datasets of the three LBSNs we analyze. The detailed analysis of the original datasets is presented in Section 6.2. Section 6.3 presents the design of LBSNSim, which is evaluated in Section 6.4. Section 6.5 concludes the chapter.

6.1 Datasets

Our datasets are from three LBSNs: Gowalla, Brightkite, and Foursquare. We consider the check-ins whose location has latitude between 24°N and 50°N, and longitude between 64°W and 126°W. This includes the mainland of the USA, where the three LBSNs have the majority of check-ins.

Gowalla is an LBSN that was founded in 2007. The dataset was collected by Cho et al. [28] over the period between Feb. 2009 and Oct. 2010, which consists of 3.7 million check-ins from 196,591 users, as well as the friendship edges between those users.

Brightkite is an LBSN launched in 2007. The dataset was collected by Cho et al. [28] between Apr. 2008 and Oct. 2010, which consists of 2.9 million check-ins from 58,228 users. Since the original friendships in Brightkite are directed,

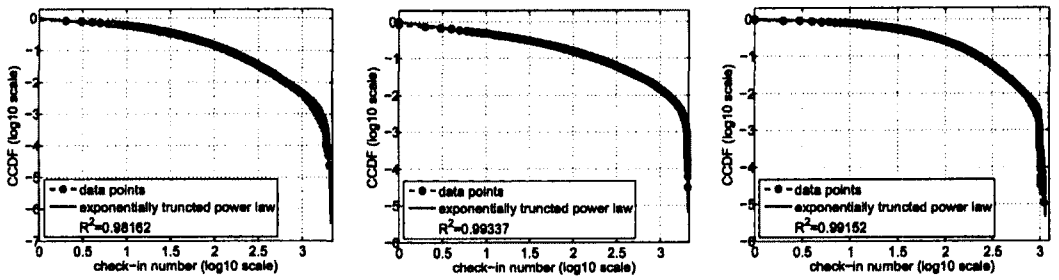
the authors constructed an undirected friendship network by only considering bi-directional edges.

Foursquare was created in 2009 and it is now the most popular LBSN with over 30 million users and 3 billion check-ins as of Jan. 2013 [1]. The dataset was collected by Cheng et al. [27] between Sep. 2010 and Jan. 2011, which consists of 8.0 million check-ins from 93,115 users. Since Foursquare does not allow unauthorized access to users' friend lists, this dataset does not contain the friendship graph.

The statistics of the three datasets are shown in Table 6.1. Each check-in in the datasets is stored as a tuple $\langle userID, time, latitude, longitude, venueID \rangle$, while each friendship edge is stored as a tuple $\langle userID_A, userID_B \rangle$. Note that check-ins at the same venue have the same GPS coordinates, provided by the corresponding LBSN.

6.2 Data Analysis

In this section, we investigate the statistical characteristics of the original datasets. We extract and analyze the following data: the number of check-ins of each user, the spatial displacement of consecutive check-ins, the temporal interval of consecutive check-ins, distance between friends, the number of friends of each user, and the number of check-ins at each venue. Our findings suggest that the datasets share many universal features, which guides the design and evaluation of LBSNSim.



(a) Gowalla (b) Brightkite (c) Foursquare

Figure 6.1: Exponentially truncated power law distribution of check-in numbers.

6.2.1 Number of check-ins

We begin with an investigation of the number of check-ins made by each user. We plot in Figure 6.1 the log-log CCDF (complementary cumulative distribution function) of the number of check-ins made by each user in the three datasets, as the CCDF plot is known to show the tail pattern of a distribution better than the PDF (probability density function) plot. All the plots exhibit a sizable downward curvature, which cannot be fitted with a straight line, indicating a significant deviation from a power law distribution. Instead, by analyzing the data, we find that they can be well fitted with an exponentially truncated power law distribution [31], whose probability density function is

$$p(x) \propto x^{-\alpha} e^{-\lambda x}, \tag{6.1}$$

where α and λ are two parameters to be estimated for each dataset. Regarding this density function, we first give a property (Lemma 1) that will be used when we formally define the distribution (Lemma 2) and estimate the parameters (Lemma 3).

Lemma 1. *Define*

$$F_x(\alpha, \lambda) = \int_x^{+\infty} t^{-\alpha} e^{-\lambda t} dt.$$

Then

$$F_x(\alpha, \lambda) = \lambda^{\alpha-1}\Gamma(1-\alpha, \lambda x),$$

where $\Gamma(a, x)$ *is the incomplete gamma function defined as*

$$\Gamma(a, x) = \int_x^{+\infty} t^{a-1}e^{-t}dt.$$

Proof. Substituting λt in $F_x(\alpha, \lambda)$ by s, we have

$$F_x(\alpha, \lambda) = \int_{\lambda x}^{+\infty} (\frac{s}{\lambda})^{-\alpha}e^{-s} \cdot \frac{1}{\lambda}ds.$$

Rearranging the terms proves the lemma. □

In practice, there exists a lower bound x_{min} and an upper bound x_{max} of the feasible x. Taking this into account, we derive the probability density function.

Lemma 2. *The probability density function for variable* $x \in [x_{\text{min}}, x_{\text{max}}]$ *satisfying equation (6.1) is as follows.*

$$p(x) = \frac{\lambda^{1-\alpha}}{\Gamma(1-\alpha, \lambda x_{\text{min}}) - \Gamma(1-\alpha, \lambda x_{\text{max}})} x^{-\alpha}e^{-\lambda x}.$$

Proof. Note that

$$\begin{aligned}\int_{x_{\text{min}}}^{x_{\text{max}}} x^{-\alpha}e^{-\lambda x}dx &= \int_{x_{\text{min}}}^{+\infty} x^{-\alpha}e^{-\lambda x}dx - \int_{x_{\text{max}}}^{+\infty} x^{-\alpha}e^{-\lambda x}dx \\ &= F_{x_{\text{min}}}(\alpha, \lambda) - F_{x_{\text{max}}}(\alpha, \lambda).\end{aligned}$$

Thus, $\int_{x_{\text{min}}}^{x_{\text{max}}} p(x)dx = 1$. □

To estimate the parameters α and λ, there are generally two methods, max-

imum likelihood estimation (MLE) and the moment method. We implemented both methods and found that they give similar results, while the moment method converges much faster. The underlying idea of the moment method is to equate the population moments with the sample moments, and solve the resulting equations. We only use the first and the second moments, since there are two parameters to be determined. The two population moments are stated in the following lemma.

Lemma 3. *For the distribution defined in Lemma 2, we have*

$$E[x] = \frac{\Gamma(2-\alpha, \lambda x_{\min}) - \Gamma(2-\alpha, \lambda x_{\max})}{\lambda(\Gamma(1-\alpha, \lambda x_{\min}) - \Gamma(1-\alpha, \lambda x_{\max}))}$$

and

$$E[x^2] = \frac{\Gamma(3-\alpha, \lambda x_{\min}) - \Gamma(3-\alpha, \lambda x_{\max})}{\lambda^2(\Gamma(1-\alpha, \lambda x_{\min}) - \Gamma(1-\alpha, \lambda x_{\max}))}.$$

Proof. We only give the derivation for $E[x]$. The derivation for $E[x^2]$ is similar.

$$\begin{aligned} E[x] &= \int_{x_{\min}}^{x_{\max}} x \cdot p(x) dx \\ &= \int_{x_{\min}}^{x_{\max}} x \cdot \frac{\lambda^{1-\alpha} x^{-\alpha} e^{-\lambda x}}{\Gamma(1-\alpha, \lambda x_{\min}) - \Gamma(1-\alpha, \lambda x_{\max})} dx \\ &= \frac{\lambda^{1-\alpha} \int_{x_{\min}}^{x_{\max}} x^{-(\alpha-1)} e^{-\lambda x} dx}{\Gamma(1-\alpha, \lambda x_{\min}) - \Gamma(1-\alpha, \lambda x_{\max})} \\ &= \frac{\Gamma(2-\alpha, \lambda x_{\min}) - \Gamma(2-\alpha, \lambda x_{\max})}{\lambda(\Gamma(1-\alpha, \lambda x_{\min}) - \Gamma(1-\alpha, \lambda x_{\max}))}, \end{aligned}$$

where the last equality can be obtained by Lemma 1 and the proof of Lemma 2.

□

Suppose d_i is user i's check-in number. With the moment method, the esti-

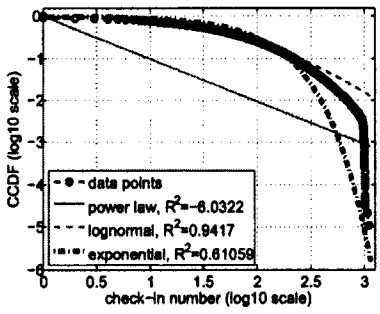
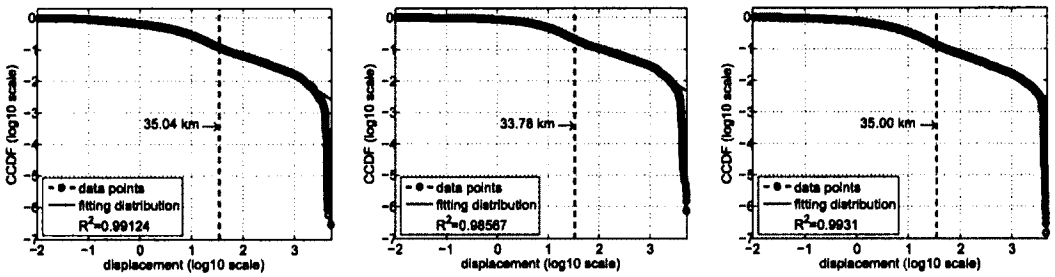


Figure 6.2: Several other fits on the Foursquare check-in number data.



(a) Gowalla (b) Brightkite (c) Foursquare

Figure 6.3: Two-segment distribution of displacements (km).

mated two parameters $\hat{\alpha}$ and $\hat{\lambda}$ are the solution to the system of equations

$$\begin{cases} E[x] = \frac{1}{n}\sum_i d_i \\ E[x^2] = \frac{1}{n}\sum_i d_i^2. \end{cases} \tag{6.2}$$

Based on the parameters estimated with the moment method, we plot the CCDF of the fitting distributions in Figure 6.1. The figure shows that the exponentially truncated power law distribution fits the data well. To further investigate the goodness-of-fit, we use the coefficient of determination of the data fit, also known as R^2, as an indicator of fitting errors. The closer R^2 is to 1, the better the distribution fits the data. As shown in Figure 6.1, the R^2 value is close to 1 for all the three datasets, indicating a good fit.

We have also tried to fit the data with other distributions. However, none of them fits the data better than the exponentially truncated power law distribution.

For example, as shown in Figure 6.2, the R^2 values of the three fits on the Foursquare dataset are all lower than the R^2 value of the exponentially truncated power law fit.

All the CCDF plots in Figure 6.1 show truncations of check-in numbers larger than several hundred, whose effects are sharp drops in the frequency of very large check-in numbers. One possible explanation of the exponential truncations is that the set of candidate venues a user can check-in is geographically constrained by factors like boundaries and physical obstructions. Furthermore, previous research has shown a check-in fatigue after prolonged use of LBSNs [68]. As a result, the frequency of very large check-in numbers in the datasets is lower than what would be in a power law distribution, whose CCDF is a straight line in the log-log scale.

6.2.2 Displacement of consecutive check-ins

In this subsection we study the spatial displacement of consecutive check-ins. We measure the distances between all the pairs of consecutive check-ins made by each user in the three datasets, and plot the log-log CCDF in Figure 6.3. It is easy to see that all the three plots consist of two segments with different curvature shapes that meet at a transition point. By analyzing the data we find that user displacements can be well fitted with an exponentially truncated power law distribution in the body, and a lognormal distribution in the tail. How to find the optimal transition point is shown as follows.

Definition 1. *Denote by x_0 a transition point. An exponentially truncated power law distribution with lognormal in the tail is defined by the following probability*

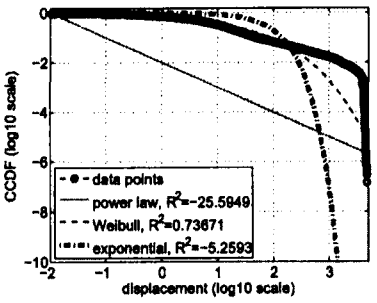
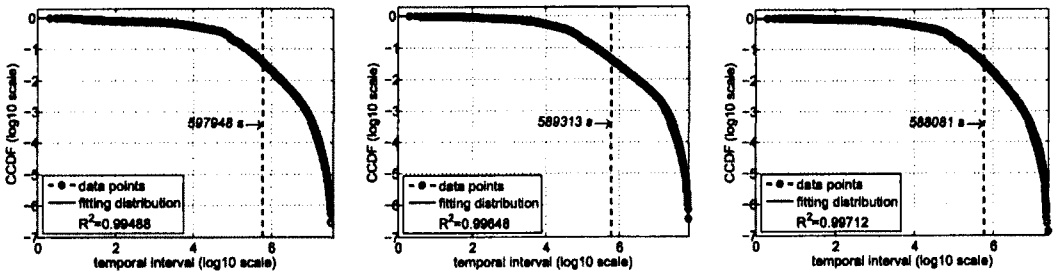


Figure 6.4: Several other fits on the Foursquare displacement (km) data.



(a) Gowalla (b) Brightkite (c) Foursquare

Figure 6.5: Two-segment distribution of temporal intervals (seconds).

density function

$$p(x) = \begin{cases} \beta \cdot \frac{\lambda^{1-\alpha}}{\Gamma(1-\alpha,\lambda x_{\min}) - \Gamma(1-\alpha,\lambda x_0)} x^{-\alpha} e^{-\lambda x} & \text{if } x \le x_0 \\ (1-\beta) \cdot \frac{1}{(x-x_0)\sigma\sqrt{2\pi}} e^{\frac{-(\ln(x-x_0)-\mu)^2}{2\sigma^2}} & \text{if } x > x_0, \end{cases}$$

where β is the probability that $x \le x_0$. Denote by $p_{x_0^-}(x)$ the probability density for $x \le x_0$, and $p_{x_0^+}(x)$ the probability density for $x > x_0$.

The transition point x_0 is estimated with MLE. Denote by $d_1, d_2, \ldots, d_n$ the displacement data. Then, for any given x_0, the likelihood $L(x_0)$ can be computed as

$$L(x_0) = \prod_{d_i \le x_0} p_{x_0^-}(d_i) \cdot \prod_{d_i > x_0} p_{x_0^+}(d_i),$$

where parameters α, λ in $p_{x_0^-}$ are estimated by our previous method on set $\{d_i \mid d_i \le x_0\}$, parameters σ, μ in $p_{x_0^+}$ are estimated by the standard routine on set

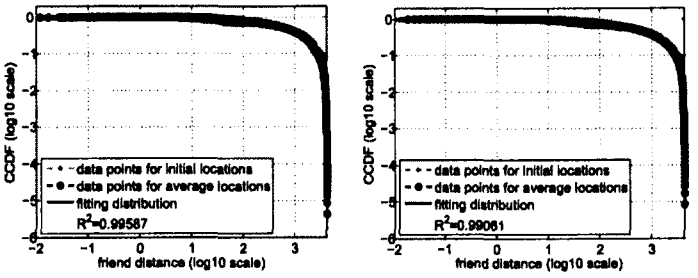
$\{d_i - x_0 \mid d_i > x_0\}$, and parameter β is simply the ratio $\frac{|\{d_i|d_i \leq x_0\}|}{|\{d_i\}|}$. The estimated x_0 is the one that maximizes $L(x_0)$.

The estimated transition points of the Gowalla, Brightkite, and Foursquare datasets are shown in Figure 6.3, which match closely and are similar to the reach of an ordinary US city. The results indicate that user inter-checkin displacements exhibit two different behaviors: Displacements within the reach of the borders of a city correspond to the daily short movements, and a vast majority of all the user displacements belong to this type, while displacements beyond the reach of the borders of a city correspond to the occasional long trips, and only a small fraction of the displacements fall into this category. Based on the estimated parameters, we plot the CCDF of the fitting distributions in Figure 6.3, which shows that the two-segment distribution is a good fit to the data. We also plot several other fits over the Foursquare displacement data in Figure 6.4. None of those distributions fits the data better than the two-segment distribution.

6.2.3 Temporal interval of consecutive check-ins

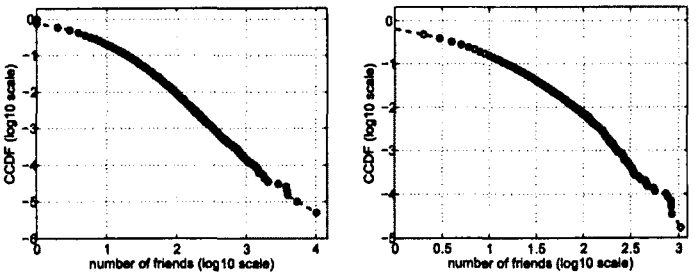
Next we study the temporal interval of consecutive check-ins. We plot in Figure 6.5 the CCDF of the temporal intervals between all the pairs of consecutive check-ins made by each user in the three datasets. Again, the plots exhibit a two-segment pattern. We find that small temporal intervals can be well fitted with an exponentially truncated power law distribution, while large temporal intervals can be well fitted with a Weibull distribution, which meet at a transition point.

We use the technique described in the previous subsection to estimate the optimal transition point. The results are shown in Figure 6.5. The transition points of all the three datasets match well, and they are close to the temporal



(a) Gowalla (b) Brightkite

Figure 6.6: Truncated Weibull distribution of friend distances (km).



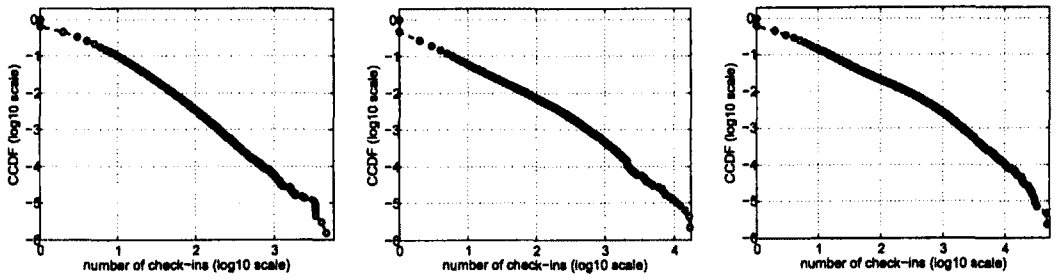
(a) Gowalla (b) Brightkite

Figure 6.7: Power law distribution of friend numbers.

length of a week in seconds (604,800s). This is a strong indicator that user check-in intervals exhibit two different patterns: the check-in intervals shorter than one week follow some weekly temporal rhythms, as shown in previous research [27, 81]. On the other hand, the check-in intervals longer than a week tend to arise from the more random check-ins made by users, e.g., when a user visits a new venue. We plot the CCDF of the fitting distributions based on the estimated parameters in Figure 6.5, which shows good fits to the original data with high R^2 values.

6.2.4 Distance between friends

In this subsection we study the geographic distance between friends in LBSNs. We consider two cases. One is we measure the distance between each pair of friends' first check-ins, i.e., their initial locations, and the other is we measure the



(a) Gowalla (b) Brightkite (c) Foursquare

Figure 6.8: Power law distribution of venue popularity.

distance between each pair of friends' average locations over all their check-ins. We plot in Figure 6.6 the CCDF for both cases. Note that since the Foursquare dataset does not contain friendship information, we only plot the CCDF figure of the Gowalla and Brightkite datasets. Figure 6.6 shows that for both datasets the initial location curve and the average location curve match closely, indicating that there is no significant difference between these two measurements.

The downward CCDF curves can be well fitted by a truncated Weibull distribution, which is obtained by restricting the random variable of Weibull distribution within the range $(0, x_{max}]$, and normalizing the probability density function accordingly. Based on the parameters estimated with MLE, we plot the CCDF of the fitting distributions in Figure 6.6. The high R^2 values indicate a good fit. For comparison, we measure the distances between randomly selected 1,000,000 pairs of arbitrary users (strangers). The average distance between friends (1,040 km) is much smaller than the average distance between strangers (2,021 km), indicating that friendship tends to be established between geographically close users in LBSNs.

6.2.5 Number of friends

The degree (number of friends) distributions of the Gowalla and Brightkite datasets are reported in Figure 6.7. The CCDF is approximately a straight line in the log-log scale, which illustrates that user degrees follow a power law distribution. This result is consistent with the previous findings on online social networks [88, 104]: the majority of users have small degrees, while a small number of users have significantly larger degrees, which are the ``hub" nodes in the social graph.

6.2.6 Venue popularity

We define the popularity of a venue as the number of check-ins made at this venue. To investigate the difference in popularity across all the venues, we plot in Figure 6.8 the CCDF of the number of check-ins at each venue in the three datasets. Again, the CCDF can be approximately fitted with a straight line in the log-log scale, indicating a power law distribution. The heavy tail of power law implies that only a few the most popular venues receive a large number of check-ins.

6.3 Modeling LBSNs

Based on our findings, in this section we build LBSNSim, a trace-driven model that generates synthetic LBSN datasets capturing the statistical features of the original datasets. We assume that the locations of a set of venues are known, which will be used as the input to our algorithm. The synthetic users' check-ins will adhere to these venues in the generated datasets. We believe that

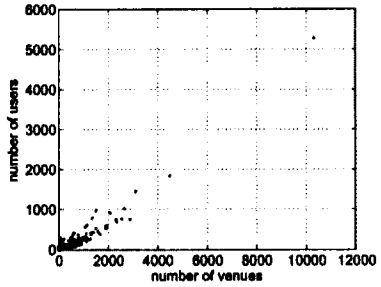


Figure 6.9: The number of users versus the number of venues in each cell.

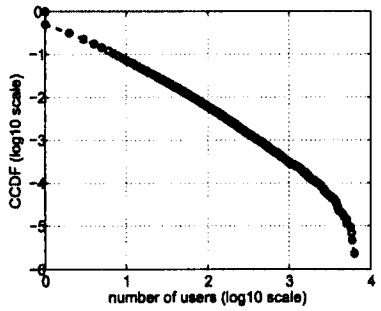


Figure 6.10: Power-law venue popularity based on the first check-in of each user in the Foursquare dataset.

this is a rational assumption, because releasing the venue information does not compromise any user privacy. Also, the information is readily available, which can be extracted from the published datasets.

The development of our model consists of three steps: (1) generating the initial location of each synthetic user; (2) building the friendship graph considering both social and geographic factors; (3) generating the check-ins of each user such that the displacements and temporal intervals between consecutive check-ins follow the found distributions.

6.3.1 Generating the initial location

To generate the initial location, i.e., the location of the first check-in, of each user in the synthetic dataset, our algorithm relies on the hypothesis that the user density is proportional to the venue density in a given area. To verify this

hypothesis, we discretize the mainland of the USA into 0.1° latitude by 0.1° longitude cells, and plot in Figure 6.9 the number of venues in each cell versus the number of users whose first check-in is in that cell, based on our Foursquare dataset. The figure signals a significant linear correlation (the correlation coefficient is 0.9324) between venue density and user density, and thus verifies our hypothesis. The other two datasets both exhibit a similar pattern. Our algorithm runs as follows.

Assume the set of venues in $cell(i,j)$ is V_{ij}, and the total number of synthetic users is n. Starting from the first to the n^{th} user, for each user, based on our finding, the probability that her initial location is in $cell(i,j)$ is $\frac{|V_{ij}|}{\sum_{i,j}|V_{ij}|}$. Suppose $cell(p,q)$ is selected. The probability of choosing venue $v \in V_{pq}$ as her initial location is proportional to $n_v + \epsilon$, i.e., $\frac{n_v+\epsilon}{\sum_{v \in V_{pq}} n_v + |V_{pq}|\epsilon}$, where n_v is the number of users who have chosen v as their initial location, and ϵ is a small constant. The plus-ϵ operation guarantees that the venues which have not been selected before still have an opportunity of being chosen.

This generation process implies the richer-get-richer property: the larger number of users that have chosen a venue as their initial location, the higher probability that the next user will select this venue as her initial location. This is consistent with the power law distribution of the number of users that have chosen each venue as their initial location in the original dataset, as shown in Figure 6.10.

6.3.2 Building the friendship graph

The second step is to build the friendship graph. As shown in the previous section, in the real datasets user degrees follow a power law distribution, while the

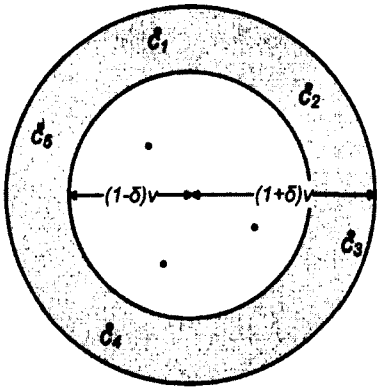


Figure 6.11: Ring area to search for the destination node.

distances between friends follow a truncated Weibull distribution. The generated friendship graph should preserve both properties.

We use an extended preferential attachment process, similar to the one proposed by Capocci et al. [22], to reproduce the power law degree distribution.

Assume the total number of friendship edges to create is e. Each user in a social graph is represented as a node. The process starts with an initial set of m_0 nodes with $m_0 > e/n$, where n is the total number of synthetic users. A clique topology is generated among those m_0 nodes, i.e., each node is linked to the other $m_0 - 1$ nodes. These are the startup nodes in the social graph. For example, they may be the administrators of the LBSN.

Starting from the $(m_0 + 1)^{th}$ user to the n^{th} user, at each step, a new node and e/n new edges are introduced into the social graph. For each new edge, with probability p, the edge connects the new node with an existing node. With probability $1 - p$, the edge originates from an existing node and ends at another existing node, and the probability of choosing an existing node i as the source node is proportional to i's degree, i.e., $\frac{d_i}{\sum_i d_i}$. In both cases, how to select the destination node is explained below.

Given a source node, to choose the destination node of a friendship edge,

we first randomly sample a distance v from the truncated Weibull fitting distribution of the distances between friends' initial locations, which is acquired in the previous section. If $v \geq \tau$, we draw a circular ring area on the map centered at the initial location of the source node, whose inner radius is $(1-\delta)v$, and outer radius is $(1+\delta)v$, where δ is a tunable parameter, as shown in Figure 6.11. All the nodes whose initial location is within this ring area (the grey area) are treated as candidate nodes, from which the destination node will be chosen. Otherwise, if $v < \tau$, all the nodes whose initial location falls into the circular area centered at the initial location of the source node and with radius v are considered as candidate nodes. Here, τ is a small threshold distance, which is set to 0.5 km in our experiments. Assume the set of candidate nodes is C. The probability of choosing C_i, the i^{th} node in C, as the destination node is proportional to its degree, i.e., $\frac{d_{C_i}}{\sum_{C_i \in C} d_{C_i}}$. If $|C| = 0$, we sample a new distance and repeat the process.

6.3.3 Generating the check-ins

The last step of our model is to generate all the check-ins of each user. Note that the location of each user's first check-in has already been generated in the first step.

As mentioned in the previous section, each check-in is composed of the location information, represented as a venue ID, and a timestamp. Our task is to generate both pieces of information for each check-in, such that the synthetic check-in traces capture the properties of the original traces.

Our algorithm runs by first assigning a check-in number to each user. For each user, we randomly sample a check-in number from the exponentially trun-

cated power law fitting distribution of user check-in numbers acquired in the previous section. To assign timestamps to these check-ins, we first need to generate the timestamp of each user's first check-in. To achieve this we sorted the timestamps of all the users' first check-ins in the real dataset in increasing order, and found that the CCDF of the temporal intervals between consecutive timestamps in this sorted list can be approximated by a power law distribution. Given the timestamp of the first user's first check-in, with the power-law parameters estimated by MLE, we are able to generate a sequence of timestamps of the other users' first check-ins, such that the temporal intervals follow the power law distribution.

Assume the timestamp of user i's first check-in is t_{c_1}, and her check-in number is n_i. To generate the timestamps of i's following check-ins, we randomly sample $n_i - 1$ temporal intervals $v_1, ..., v_{n_i-1}$ from the two-segment fitting distribution of the temporal intervals between consecutive check-ins made by users, which is acquired in the previous section. The timestamp of the j^{th} check-in made by user i is thus: $t_{c_j} = t_{c_1} + \sum_{k=1}^{j-1} v_k$.

Now we have generated the timestamps of all the check-ins, and our next task is to generate the location of each check-in, i.e., to determine which venue the check-in adheres to. We achieve this by first sorting all the check-ins by their timestamps in increasing order. Starting from the first check-in to the last check-in, for each encountered check-in c, we check if c is its creator u's first check-in. If so, then c's location is the same as u's initial location. Otherwise, we randomly sample a displacement v from the two-segment fitting distribution of the displacements between consecutive check-ins made by users. Assume c is u's i^{th} check-in, where $i > 1$. If $v \geq \tau$, we draw a circular ring area on the map centered at the location of u's $(i-1)^{th}$ check-in, whose inner radius is $(1-\delta)v$,

and outer radius is $(1+\delta)v$. All the venues falling into this ring area are treated as candidate venues that c may adhere to. If $v < \tau$, we consider all the venues falling into the circular area centered at the location of u's $(i-1)^{th}$ check-in and with radius v as candidate venues.

Assume the set of candidate venues is V. We consider two cases when determining c's venue. Let V_f be the set of venues in V that have been previously checked-in by some of u's friends, and $V_s = V - V_f$. If both V_f and V_s are not empty, then with probability p', c adheres to a venue in V_f. The probability that c adheres to venue i in V_f is proportional to the number of times that u's friends have previously checked-in at this venue, i.e., $\frac{f_i}{\sum_{i \in V_f} f_i}$. With probability $1-p'$, c adheres to a venue in V_s. The probability that c adheres to venue i in V_s is proportional to ϵ plus the number of times that the non-friend users (strangers) have previously checked-in at this venue, i.e., $\frac{s_i+\epsilon}{\sum_{i \in V_s} s_i + |V_s|\epsilon}$. The plus-$\epsilon$ operation guarantees that the venues which have not been checked-in by any user before still have an opportunity of being chosen. By introducing the parameter p', we take into account the difference between check-ins made by friends and by strangers. This captures the social influence on users' check-in behavior, which was found in previous research [28, 40]. If either V_f or V_s is empty, then we only consider the venues in the non-empty set. If both V_f and V_s are empty, then we sample a new distance and repeat the process.

Note that in the original datasets we do not observe obvious correlation between the displacement and the temporal interval of consecutive check-ins. The correlation coefficients of the Gowalla, Brightkite, and Foursquare datasets are 0.1306, 0.0972, and 0.1256, respectively. This finding is rational because check-in is a spontaneous user behavior, which is different from continuous location sensing. It may take a long time for a user to make two check-ins, while

the displacement between them may be very small. Therefore, when building the model we do not consider this correlation. Instead, we only filter out the generated consecutive check-ins which imply an unrealistically high transit speed.

6.4 Model Verification

In this section, we evaluate the fidelity of LBSNSim by verifying whether LBSNSim can generate synthetic LBSN datasets that capture the statistical features observed in the original datasets.

6.4.1 Experimental Setup

In the evaluation we use the Gowalla dataset as the target dataset. The parameters of the distributions used in our model are estimated based on this dataset. The number of users in the model-generated datasets is 100,000, the number of friendship edges is 500,000, and the total number of check-ins is 5,000,000. The other parameters used in the model are experimentally set as: $m_0 = 20, p = 0.4, p' = 0.8, \delta = 0.25, \epsilon = 1, \tau = 0.5$ km. We use the venues extracted from the Gowalla dataset as the input to our algorithm. Note that using LBSNSim, researchers can generate LBSN datasets with different scales and properties by tuning the parameters in the model, and experiment with these datasets to produce statistically confident results.

6.4.2 Evaluation Results

In this section we test a wide range of statistical features, including distance between friends, radius of gyration (explained later), temporal intervals consid-

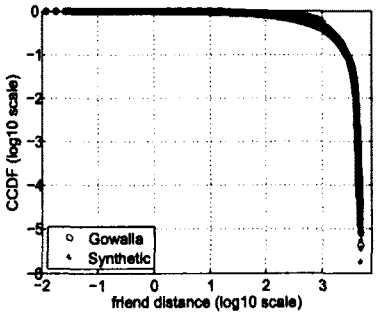


Figure 6.12: Friend distances (km) based on average locations.

ering all the check-ins, node degrees, venue popularity, and social influence on check-ins. We do not test the features that are explicitly modeled by LBSNSim, including the truncated power law check-in number distribution, and the two-segment distributions of the temporal intervals and displacements between consecutive check-ins made by each user. In the experiments we generate 50 realizations of our model and we observe no significant difference among them in the statistical features we evaluate, so for each feature we compare the target dataset with a randomly selected model-generated dataset. In the application-level test, we run an LBSN-based application with both the target dataset and the synthetic datasets as input, and compare their results to quantify the fidelity of LBSNSim.

Distance between friends. In the previous section, we use distance as a constraint on friendship generation, by sampling a distance v from the truncated Weibull fitting distribution of the distances between friends' initial locations each time when a new friendship edge is created, and using it to search for the destination node of the edge. However, the resulting distance between the source node and the destination node is only an approximation of v, as the destination node is chosen from all the nodes falling into the ring area with inner radius $(1-\delta)v$ and outer radius $(1+\delta)v$. As shown in Figure 6.12, even with this ap-

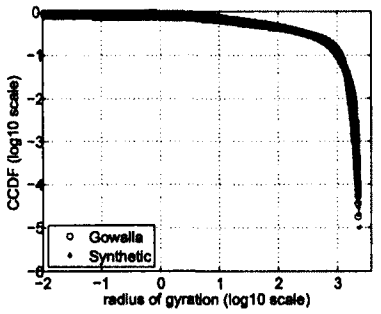


Figure 6.13: Radius of gyration (km).

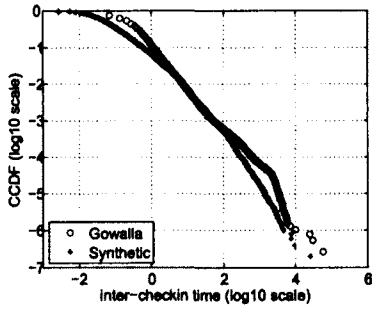


Figure 6.14: Inter-checkin time (seconds) of all the check-ins, rescaled by dividing by the average time interval.

proximation, the distribution of the distances between friends' average locations in the model-generated dataset still matches well with that in the target dataset.

Radius of gyration. The radius of gyration of a user is defined as the root mean square distance of a user's check-ins from their center of mass:

$$R_g = \sqrt{\frac{1}{n_i}\sum_{j=1}^{n_i}(distance(c_j, c_a))^2},$$

where n_i is user i's check-in number, c_j is the location of the j^{th} check-in, and c_a is the average location over all the check-ins. Radius of gyration measures the ``spread" of a user's check-ins: the larger radius of gyration is, the more widely a user's check-ins are dispersed. Figure 6.13 compares the distribution of radius of gyration in the model-produced dataset with that extracted from the target dataset, which shows that the two distributions match well.

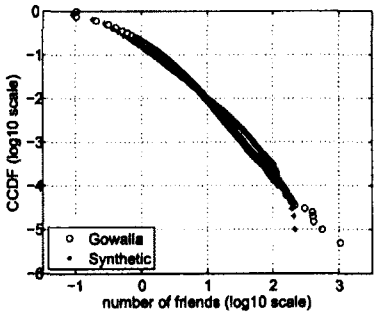


Figure 6.15: Number of friends of each user, rescaled by dividing by the average friend number.

Inter-checkin time of all the check-ins. To measure the distribution of the inter-checkin time considering all the check-ins, we sort all the user check-ins by their timestamps in increasing order. Figure 6.14 plots the distribution of the temporal interval between every pair of consecutive check-ins in this sorted list, for both the synthetic dataset and the target dataset. The CCDFs of both distributions can be approximated by a straight line in the log-log scale, which indicates a power law distribution, and they also match closely. Note that this distribution is different from the distribution of the temporal intervals between consecutive check-ins made by each user, which is studied in the data analysis section.

Number of friends. Figure 6.15 compares the distribution of the friend numbers in the model-produced dataset with that in the target dataset. Both CCDF curves are approximately a straight line in the log-log scale with similar slope. This validates the correctness of our extended preferential attachment process used to generate the friendship graph.

Venue popularity. In Figure 6.16 we plot the distribution of venue popularity, defined as the number of check-ins at each venue, in the generated dataset and in the target dataset. The distributions are rescaled by dividing by the average venue popularity. The figure illustrates that our check-in generation method

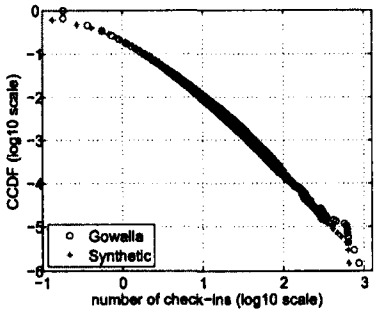


Figure 6.16: Venue popularity, rescaled by dividing by the average venue popularity.

Table 6.2: False positive and negative rates of SybilDefender

	Original		Synthetic	
	F^+	F^-	F^+	F^-
2000 sybil nodes	0.3%	0.2%	0.4%	0.2%
1000 sybil nodes	0.3%	0.4%	0.3%	0.5%
500 sybil nodes	0.2%	0.6%	0.2%	0.8%

captures the power-law venue popularity found in the original datasets.

Social influence on check-ins. To quantify the social influence on users' check-in behavior, we measure the probability that two friends have checked-in at the same venue, and compare it with the probability that two strangers have checked-in at the same venue. The probability is measured by randomly selecting 100,000 pairs of users and counting the number of pairs that have checked-in at at least one common venue. All the results are averaged over 50 runs. The friend-case probability and the stranger-case probability for the Gowalla dataset are 16.29% and 0.39%, respectively. For the model-generated dataset they are 14.50% and 0.59%, respectively. The results demonstrate that similar to the real datasets, the model-generated datasets exhibit strong social influence on users' check-in behavior, i.e., people are more likely to visit places that their friends visited in the past.

Application-level test. In this subsection, we compare the results of an

LBSN-based application obtained by using the target dataset with those obtained by using the synthetic datasets. SybilDefender was proposed in previous research to defend against sybil attacks in social networks [102], when an attacker creates many bogus identities to compromise the running of the system. To extend SybilDefender to LBSNs, we augment the sybil identification algorithm by considering edge weights, which are defined as the number of venues that have been checked-in by both ending nodes of a friendship edge. The algorithm runs by performing weighted random walks in the friendship graph. At each step of a weighted random walk, edge weights are considered when choosing the next hop. The weighted sybil identification algorithm takes the target dataset and the synthetic datasets as input. In each experiment we randomly create sybil nodes forming a connected scale-free topology, with a small number of edges linking to the largest connected component of the friendship graph. The results are averaged over 50 runs.

Table 6.2 shows the average false positive and negative rates of the weighted sybil identification algorithm, when running on the target dataset and on the synthetic datasets. It demonstrates that the application-level results obtained by using the synthetic datasets closely match those obtained by using the original dataset.

6.5 Conclusion

In this chapter, we analyze the statistical features extracted from the data of three LBSNs, and propose LBSNSim, a trace-driven model for generating synthetic LBSN datasets that capture the properties of the original datasets. Evaluation shows that the synthetic datasets generated by LBSNSim are suffi-

ciently representative of real-world LBSN datasets in a wide range of statistical features, and that high fidelity results can be produced using the synthetic datasets in the application-level test. This verifies the feasibility of using the model-generated datasets as replacements for real LBSN datasets.

Chapter 7

Conclusion

With OSNs central to so many peoples's lives, it is critical to address the rising tension between the value of participation and the security and privacy threats to OSN users. On the one hand, through OSNs users can establish and strengthen social connections, communicate with one another, and maintain personal data. On the other hand, users' privacy and security will be compromised if their personal data are leaked or they are targeted by various attacks on OSNs. To improve security and privacy in OSNs, in this dissertation, we address four challenging issues of OSNs. First, we propose Fast Mencius, a crash-fault tolerant state machine replication protocol that has low commit latency and high throughput in wide-area networks. Second, we propose SybilDefender, a sybil defense mechanism that leverages the social network topologies to detect sybil nodes. Third, we propose MobiShare, a privacy-preserving location-sharing scheme that supports sharing locations between both friends and strangers without compromising user privacy. Fourth, we propose LBSNSim, a trace-driven LBSN model for generating synthetic LBSN datasets that can be used in place of the real LBSN datasets.

Correspondingly, we have made four main contributions in this dissertation. First, we have improved the infrastructure reliability of OSNs with our crash-fault tolerant state machine replication protocol. Our protocol enhances the state of the art with mechanisms that allow fast replicas to proceed without being delayed by the slowest replica, and thus achieves much smaller commit latency. Second, we have strengthened the security of OSNs against sybil attacks with our sybil defense mechanism. Compared with previous work, our sybil defense mechanism is significantly more efficient and effective in detecting sybil nodes by creatively leveraging the coverage difference between random walks in the honest region and in the sybil region. Third, we have contributed to protecting user privacy on OSNs with our privacy-preserving location-sharing scheme. With this scheme, we show a new direction to preserve OSN users' location privacy by separating location data from user identities, which addresses the weaknesses of previous encryption and cloaking schemes. Finally, we have mitigated the risk of privacy leakage from releasing original LBSN datasets with our LBSN model, which can generate synthetic LBSN datasets. To the best of our knowledge, this is the first time that detailed distributions have been found and explained for a wide range of LBSN features, and this is also the first time that an evolution model has been proposed for LBSNs. Overall, we have addressed a representative security/privacy issue for each of the OSN components, and combining our work serves to build more secure and privacy-preserving OSNs.

During our studies we have gained experience that is helpful to the general advancement of research in this area. First, we find that a multi-leader design is advantageous to improving performance when solving consensus problems. The reason is that the multi-leader design eliminates the performance

bottleneck incurred by a single leader, and it can fully utilize the available bandwidth and computation power of all the servers. Second, we find that social graph topologies are valuable resources that can be used to address security issues related to social connections, since these topologies exhibit the structured relationships between human beings. Besides the sybil attack we have studied before, other examples of such security issues include fishing attack, de-anonymization attack, spam, and false recommendations/reviews. Third, we find that the separation approach is a promising way to address privacy issues in OSNs. The basic idea is to separately store and process users' sensitive information and their identities, such that the adversaries cannot link user identities to the protected data.

As for future work, we believe that how to improve the infrastructure reliability and how to prevent the functioning of OSNs from being disturbed by malicious behaviors remain hot research topics, and there is still plenty of room for improvement. For the infrastructure layer of OSNs, the infrastructure reliability may be compromised not only by crash failures, but also by Byzantine failures, in which the servers hosting OSNs are controlled by adversaries and behave arbitrarily [23]. Byzantine failures are harder to handle than crash failures, since the faulty servers can be arbitrarily malicious. We intend to investigate how to tolerate Byzantine failures in state machine replication systems with a rotating-leader design, which has the potential of achieving higher throughput than the single-leader schemes. For the function layer of OSNs, an unsolved attack that seriously threatens the security of OSN users is the spoofing attack, in which an attacker masquerades as another user and gains an illegitimate advantage [17]. Because of the hardness to verify user identities, spoofing attacks can be easily launched on OSNs without being detected. Our next step is to study how to

defend against this attack on OSNs. Finally, for the user data stored on OSNs, one problem we have not considered in this dissertation is how to ensure that the data provided by users have high quality. It has been found that the users' check-in data on OSNs can be faked [49], which will compromise the operation of all the location-based features provided by OSNs. Although some preliminary attempts have been made to address this issue [114], no concrete solution has been proposed so far. To preserve authenticity of the check-in data collected by OSNs, we plan to investigate how to detect the fake check-ins generated by malicious users.

References

[1] About foursquare. https://foursquare.com/about.

[2] Amazon elastic compute cloud (amazon ec2). http://aws.amazon.com/ec2.

[3] App store, hacked. http://thenextweb.com/apple/2010/07/04/app-store-hacked.

[4] Chronology of data breaches and security breaches 2005-present. http://www.privacyrights.org/data-breach.

[5] Enhanced 9-1-1 wireless services. http://www.fcc.gov/pshs/services/911-services/enhanced911/.

[6] Facebook statistics. http://newsroom.fb.com/content/default.aspx?NewsAreaId=22.

[7] The top 500 sites on the web. http://www.alexa.com/topsites.

[8] Trust network datasets. http://www.trustlet.org/wiki/Datasets.

[9] M. Abd-El-Malek, G. R. Ganger, G. R. Goodson, M. Reiter, and J. J. Wylie. Fault-scalable Byzantine fault-tolerant services. In *SOSP*, 2005.

[10] Y.-Y. Ahn, S. Han, H. Kwak, S. Moon, and H. Jeong. Analysis of topological characteristics of huge online social networking services. In *WWW*, 2007.

[11] R. Albert and A. Barabási. Statistical mechanics of complex networks. *Rev. Mod. Phys*, 74:47--97, 2002.

[12] L. Backstrom, E. Sun, and C. Marlow. Find me if you can: Improving geographical prediction with social and spatial proximity. In *WWW*, 2010.

[13] R. Baden, A. Bender, N. Spring, B. Bhattacharjee, and D. Starin. Persona: an online social network with user-defined privacy. In *SIGCOMM*, 2009.

[14] L. Barkhuus and A. K. Dey. Location-based services for mobile telephony: a study of users' privacy concerns. In *INTERACT*, 2003.

[15] R. Bazzi and G. Konjevod. On the establishment of distinct identities in overlay networks. In *ACM PODC*, 2005.

[16] A. R. Beresford and F. Stajano. Location privacy in pervasive computing. *IEEE Pervasive Computing*, 2(1):46--55, 2003.

[17] L. Bilge, T. Strufe, D. Balzarotti, and E. Kirda. All your contacts are belong to us: automated identity theft attacks on social networks. In *WWW*, 2009.

[18] D. Brockmann, L. Hufnagel, and T. Geisel. The scaling laws of human travel. *Nature*, 439(7075):462--465, 2006.

[19] C. Brown, V. Nicosia, S. Scellato, A. Noulas, and C. Mascolo. The importance of being placefriends: Discovering location-focused online communities. In *WOSN*, 2012.

[20] M. Burrows. The chubby lock service for loosely coupled distributed systems. In *USENIX OSDI*, pages 335--350, 2006.

[21] J.A. Calandrino, A. Kilzer, A. Narayanan, E.W. Felten, and V. Shmatikov. ``you might also like": privacy risks of collaborative filtering. In *IEEE Symposium on Security and Privacy*, 2011.

[22] A. Capocci, V. D. P. Servedio, F. Colaiori, L. S. Buriol, D. Donato, S. Leonardi, and G. Caldarelli. Preferential attachment in the growth of social networks: The internet encyclopedia wikipedia. *Physics Review E*, 74, 2006.

[23] M. Castro and B. Liskov. Practical Byzantine fault tolerance. In *USENIX OSDI*, 1999.

[24] T. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225--267, 1996.

[25] B. Charron-Bost and A. Schiper. Improving Fast Paxos: being optimistic with no overhead. In *PRDC*, 2006.

[26] R. Chen, A. Reznichenko, P. Francis, and J. Gehrke. Towards statistical queries over distributed private user data. In *USENIX NSDI*, 2012.

[27] Z. Cheng, J. Caverlee, K. Lee, and D. Z. Sui. Exploring millions of footprints in location sharing services. In *ICWSM*, 2011.

[28] E. Cho, S. A. Myers, and J. Leskovec. Friendship and mobility: User movement in location-based social networks. In *KDD*, 2011.

[29] B. Chun, P. Maniatis, S. Shenker, and J. Kubiatowicz. Attested append-only memory: Making adversaries stick to their word. In *SOSP*, 2007.

[30] H. Chun, H. Kwak, Y. H. Eom, Y. Y. Ahn, S. Moon, and H. Jeong. Comparison of online social relations in volume vs interaction: a case study of cyworld. In *ACM/USENIX IMC*, 2008.

[31] A. Clauset, C. R. Shalizi, and M. E. J. Newman. Power-law distributions in empirical data. *SIAM Review*, 51(4):661--703, 2009.

[32] L. P. Cox, A. Dalton, and V. Marupadi. Smokescreen: Flexible privacy controls for presence-sharing. In *ACM MobiSys*, 2007.

[33] G. Danezis and P. Mit. Sybilinfer: Detecting sybil nodes using social networks. In *NDSS*, 2009.

[34] D. Dobre, M. Majuntke, M. Serafini, and N. Suri. HP: Hybrid Paxos for WANs. In *EDCC*, 2010.

[35] J. R. Douceur. The sybil attack. In *IPTPS*, 2002.

[36] N. Eagle and A. Pentland. Social serendipity: Mobilizing social software. *IEEE Pervasive Computing*, 4(2):28--34, 2005.

[37] P. Erdös and A. Rényi. On random graphs. *Publicationes Mathemticae (Debrecen)*, 6:290--297, 1959.

[38] L. Fang and K. LeFevre. Privacy wizards for social networking sites. In *WWW*, 2010.

[39] M. J. Fischer, N. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374--382, 1985.

[40] H. Gao, J. Tang, and H. Liu. Exploring social-historical ties on location-based social networks. In *ICWSM*, 2012.

[41] B. Gedik and L. Liu. A customizable k-anonymity model for protecting location privacy. In *IEEE ICDCS*, 2005.

[42] M. Girvan and M. E. J. Newman. Community structure in social and biological networks. *National Academy of Sciences of the United States of America*, 99(12):7821--7826, 2002.

[43] P. Golle and K. Partridge. On the anonymity of home/work location pairs. In *Pervasive*, 2009.

[44] M. C. Gonzalez, C. A. Hidalgo, and A.-L. Barabasi. Understanding individual human mobility patterns. *Nature*, 453(7196):779--782, 2008.

[45] M. Gruteser and D. Grunwald. Anonymous usage of location-based services through spatial and temporal cloaking. In *ACM MobiSys*, 2003.

[46] M. Gruteser and B. Hoh. On the anonymity of periodic location samples. In *Second International Conference on Security in Pervasive Computing*, 2005.

[47] S. Guha, M. Jain, and V. N. Padmanabhan. Koi: A location-privacy platform for smartphone apps. In *USENIX NSDI*, 2012.

[48] P. Gundecha, G. Barbier, and H. Liu. Exploiting vulnerability to secure user privacy on a social networking site. In *KDD*, 2011.

[49] W. He, X. Liu, and M. Ren. Location cheating: A security challenge to location-based social network services. In *ICDCS*, 2011.

[50] S. Hemminger. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, pages 299--319, 1990.

[51] B. Hoh, M. Gruteser, H. Xiong, and A. Alrabady. Enhancing security and privacy in traffic-monitoring systems. *IEEE Pervasive Computing*, 5(4):38--46, 2006.

[52] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: Wait-free coordination for Internet-scale systems. In *USENIX ATC*, 2010.

[53] M. Hurfin, I. Moise, and J. L. Narzul. An adaptive Fast Paxos for making quick everlasting decisions. In *AINA*, 2011.

[54] J. J. Pfeiffer III, T. La Fond, S. Moreno, and J. Neville. Fast generation of large scale social networks while incorporating transitive closures. In *SocialCom*, 2012.

[55] T. Jagatic, N. Johnson, M. Jakobsson, and F. Menczer. Social phishing. *Communications of the ACM*, 5(10):94--100, 2007.

[56] F. P. Junqueira, Y. Mao, and K. Marzullo. Classic Paxos vs. Fast Paxos: Caveat emptor. In *HotDep*, 2007.

[57] R. Kannan, S. Vempala, and A. Vetta. On clusterings: Good, bad and spectral. In *FOCS*, 2000.

[58] H. Kido, Y. Yanagisawa, and T. Satoh. An anonymous communication technique using dummies for location-based services. In *ICPS*, 2005.

[59] M. Kim and J. Leskovec. Modeling social networks with node attributes using the multiplicative attribute graph model. In *UAI*, 2011.

[60] V. Kostakos, J. Venkatanathan, B. Reynolds, N. Sadeh, E. Toch, S. A. Shaikh, and S. Jones. Who's your best friend?: targeted privacy attacks in location-sharing social networks. In *UbiComp*, 2011.

[61] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. Zyzzyva: Speculative Byzantine fault tolerance. In *SOSP*, 2007.

[62] B. Krishnamurthy and C. E. Wills. Privacy leakage in mobile online social networks. In *WOSN*, 2010.

[63] J. Krumm. Inference attacks on location tracks. In *Pervasive*, 2007.

[64] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133--169, 1998.

[65] L. Lamport. Paxos made simple. *ACM SIGACT News*, 32(4):18--25, 2001.

[66] L. Lamport. Generalized consensus and Paxos. Technical Report MSR-TR-2005-33, Microsoft Research, 2005.

[67] L. Lamport. Fast Paxos. *Distributed Computing*, 19(2):79--103, 2006.

[68] J. Lindqvist, J. Cranshaw, J. Wiese, J. Hong, and J. Zimmerman. I'm the mayor of my house: Examining why people use Foursquare - a social-driven location sharing application. In *SIGCHI*, 2011.

[69] B. Loo, T. Condie, J. Hellerstein, P. Maniatis, T. Roscoe, and I. Stoica. Implementing declarative overlays. In *SOSP*, 2005.

[70] J. Manweiler, R. Scudellari, and L. P. Cox. Smile: Encounter-based trust for mobile social services. In *ACM CCS*, 2009.

[71] Y. Mao, F. P. Junqueira, and K. Marzullo. Mencius: Building efficient replicated state machines for WANs. In *USENIX OSDI*, 2008.

[72] Y. Mao, F. P. Junqueira, and K. Marzullo. Towards low latency state machine replication for uncivil wide-area networks. In *HotDep*, 2009.

[73] A. Mislove, M. Marcon, K. P. Gummadi, P. Druschel, and B. Bhattacharjee. Measurement and analysis of online social networks. In *ACM/USENIX IMC*, 2007.

[74] M. Mitzenmacher and E. Upfal. *Probability and Computing*. Cambridge University Press, 2005.

[75] A. Mohaisen, N. Hopper, and Y. Kim. Keep your friends close: Incorporating trust into social network-based sybil defenses. In *IEEE INFOCOM*, 2011.

[76] A. Narayanan and V. Shmatikov. De-anonymizing social networks. In *IEEE Symposium on Security and Privacy*, 2009.

[77] A. Narayanan, N. Thiagarajan, M. Lakhani, M. Hamburg, and D. Boneh. Location privacy via private proximity testing. In *NDSS*, 2011.

[78] M. E. J. Newman. The structure of scientific collaboration networks. *National Academy of Sciences of the United States of America*, 98(2):404--409, 2001.

[79] T. S. E. Ng and H. Zhang. Predicting internet network distance with coordinates-based approaches. In *IEEE INFOCOM*, 2002.

[80] A. Noulas, S. Scellato, N. Lathia, and C. Mascolo. A random walk around the city: New venue recommendation in location-based social networks. In *SocialCom*, 2012.

[81] A. Noulas, S. Scellato, C. Mascolo, and M. Pontil. An empirical study of geographic user activity patterns in foursquare. In *ICWSM*, 2011.

[82] E. Novak and Q. Li. A survey of security and privacy research in online social networks. Technical Report WM-CS-2012-02, College of William and Mary, 2012.

[83] R. S. Peterson and E. G. Sirer. Antfarm: Efficient content distribution with managed swarms. In *NSDI*, 2009.

[84] R. A. Popa, A. J. Blumberg, H. Balakrishnan, and F. H. Li. Privacy and accountability for location-based aggregate statistics. In *ACM CCS*, 2011.

[85] K. P. N. Puttaswamy and B. Y. Zhao. Preserving privacy in location-based mobile social applications. In *HotMobile*, 2010.

[86] I. Rhee, M. Shin, S. Hong, K. Lee, and S. Chong. On the Levy-walk nature of human mobility. In *IEEE INFOCOM*, 2008.

[87] H. Rowaihy, W. Enck, P. McDaniel, and T. LaPorta. Limiting sybil attacks in structured peer-to-peer networks. In *IEEE INFOCOM*, 2007.

[88] A. Sala, L. Cao, C. Wilson, R. Zablit, H. Zheng, and B. Y. Zhao. Measurement-calibrated graph models for social network experiments. In *WWW*, 2010.

[89] A. Sala, X. Zhao, C. Wilson, H. Zheng, and B. Y. Zhao. Sharing graphs using differentially private graph models. In *IMC*, 2011.

[90] S. Scellato, A. Noulas, R. Lambiotte, and C. Mascolo. Socio-spatial properties of online location-based social networks. In *ICWSM*, 2011.

[91] S. Scellato, A. Noulas, and C. Mascolo. Exploiting place features in link prediction on location-based social networks. In *KDD*, 2011.

[92] J. Scheck. Stalkers exploit cellphone gps. Wall Street Journal, 2010.

[93] B. Schilit, J. Hong, and M. Gruteser. Wireless location privacy protection. *IEEE Computer*, 36(12):135--137, 2003.

[94] R. Shokri, G. Theodorakopoulos, J. L. Boudec, and J. Hubaux. Quantifying location privacy. In *IEEE Symposium on Security and Privacy*, 2011.

[95] A. Singh, T. Das, P. Maniatis, P. Druschel, and T. Roscoe. BFT protocols under fire. In *USENIX NSDI*, 2008.

[96] F. Stutzman and J. K. Duffield. Friends only: examining a privacy-enhancing behavior in facebook. In *CHI*, 2010.

[97] J. Teng, B. Zhang, X. Li, X. Bai, and D. Xuan. E-shadow: Lubricating social interaction using mobile phones. In *IEEE ICDCS*, 2011.

[98] N. Tran, J. Li, L. Subramanian, and S. S.M. Chow. Optimal sybil-resilient node admission control. In *IEEE INFOCOM*, 2011.

[99] G. S. Veronese, M. Correia, A. N. Bessani, and L. C. Lung. EBAWA: Efficient Byzantine agreement for wide-area networks. In *HASE*, 2010.

[100] B. Viswanath, A. Mislove, M. Cha, and K. P. Gummadi. On the evolution of user interaction in facebook. In *WOSN*, 2009.

[101] B. Viswanath, A. Post, K. P. Gummadi, and A. Mislove. An analysis of social network-based sybil defenses. In *SIGCOMM*, 2010.

[102] W. Wei, F. Xu, C. C. Tan, and Q. Li. Sybildefender: Defend against sybil attacks in large social networks. In *IEEE INFOCOM*, 2012.

[103] Wei Wei, Fengyuan Xu, Chiu C. Tan, and Qun Li. Sybildefender: Defend against sybil attacks in large social networks. In *IEEE INFOCOM*, 2012.

[104] C. Wilson, B. Boe, A. Sala, K. P. N. Puttaswamy, and B. Y. Zhao. User interactions in social networks and their implications. In *EuroSys*, 2009.

[105] G. Wondracek, T. Holz, E. Kirda, and C. Kruegel. A practical attack to de-anonymize social network users. In *IEEE Symposium on Security and Privacy*, 2010.

[106] L. Xu, S. Chainan, H. Takizawa, and H. Kobayashi. Resisting sybil attack by social network and network clustering. In *SAINT*, 2010.

[107] T. Xu and Y. Cai. Exploring historical location data for anonymity preservation in location-based services. In *IEEE INFOCOM*, 2008.

[108] T. Xu and Y. Cai. Feeling-based location privacy protection for location-based services. In *ACM CCS*, 2009.

[109] M. Yabandeh, L. Franco, and R. Guerraoui. One acceptor is enough. Technical Report LPD-REPORT-2010-001, EPFL.

[110] Z. Yang, B. Zhang, J. Dai, A. Champion, D. Xuan, and D. Li. E-smalltalker: A distributed mobile system for social networking in physical proximity. In *IEEE ICDCS*, 2010.

[111] H. Yu, P. B. Gibbons, M. Kaminsky, and F. Xiao. Sybillimit: A near-optimal social network defense against sybil attacks. In *IEEE Symposium on Security and Privacy*, 2008.

[112] H. Yu, M. Kaminsky, P. B. Gibbons, and A. Flaxman. Sybilguard: defending against sybil attacks via social networks. In *SIGCOMM*, 2006.

[113] H. Yu, M. Kaminsky, P. B. Gibbons, and A. Flaxman. Sybilguard: Defending against sybil attacks via social networks. Technical Report IRP-TR-06-01, Intel Research Pittsburgh, 2006.

[114] K. Zhang, K. Pelechrinis, and P. Krishnamurthy. Detecting fake check-ins in location-based social networks through honeypot venues. In *HotMobile*, 2013.

[115] G. Zhong, I. Goldberg, and U. Hengartner. Louis, Lester and Pierre: Three protocols for location privacy. In *Privacy Enhancing Technologies*, 2007.

VITA

Wei Wei received his Bachelor of Science degree and Master of Science degree in Computer Science from Beihang University, China, in 2005 and 2008, respectively. From August 2008, he began perusing his Doctor of Philosophy degree under the supervision of Dr. Qun Li. His research interests include online social networks and consensus protocols of distributed systems.